Department of Computer Science
Series of Publications A
Report A-2014-5

Collaborative Mobile Energy Awareness

Eemil Lagerspetz

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public examination in Auditorium A129, Chemicum building, Kumpula, Helsinki on November 24^{th}, 2014 at 12 o'clock noon.

University of Helsinki
Finland

Supervisor
Sasu Tarkoma, University of Helsinki, Finland

Pre-examiners
Jukka Riekki, University of Oulu, Finland
Cecilia Mascolo, University of Cambridge, United Kingdom

Opponent
Sumi Helal, University of Florida, United States of America

Custos
Sasu Tarkoma, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/
Telephone: +358 9 1911, telefax: +358 9 191 51120

ISSN 1238-8645
ISBN 978-951-51-0460-1 (paperback)
ISBN 978-951-51-0461-8 (PDF)
Computing Reviews (1998) Classification: B.8, B.8.2, H.1.2
Helsinki 2014
Unigrafia

Collaborative Mobile Energy Awareness

Eemil Lagerspetz

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
eemil.lagerspetz@cs.helsinki.fi
http://www.cs.helsinki.fi/u/lagerspe

PhD Thesis, Series of Publications A, Report A-2014-5
Helsinki, November 2014, 60+46 pages
ISSN 1238-8645
ISBN 978-951-51-0460-1 (paperback)
ISBN 978-951-51-0461-8 (PDF)

Abstract

We have created a mobile energy measurement application and gathered energy measurement data from over 725,000 devices, running over 300,000 applications, in heterogeneous environments, and constructed models of what is normal in each context for each application. We have used this data to find energy abnormalities in the wild, and provide users of our application advice on how to deal with them. These abnormalities cannot be discovered in laboratory conditions due to the rich interaction of the smartphone and its operating environment. Employing a collaborative mobile energy-awareness application with thousands of users allows us to gather a large amount of data in a short time. Such a large and diverse dataset has helped us answer many research questions. Our work is the first collaborative approach in the area of mobile energy debugging. Information received from each device running our application improves the advice given to other users running the same applications.

The author has developed a context data gathering hub for smartphones, discovered the need for a common API that unifies network connectivity, energy awareness, and user experience, and investigated the impact of mobile collaborative energy awareness applications, to find previously unknown energy bugs on smartphones, and to improve users' knowledge of smartphone energy behavior.

Computing Reviews (1998) Categories and Subject Descriptors:
B.8 Performance and Reliability
B.8.2 Performance Analysis and Design Aids
H.1.2 User/Machine Systems, Human factors

General Terms:
Ph.D. thesis, mobile computing, energy awareness, battery, collaborative

Additional Key Words and Phrases:
context awareness, power saving, user behavior

Acknowledgements

I extend my appreciation and thanks to my supervisor, professor Sasu Tarkoma, and professor Ion Stoica, who has supported the work on Carat since the beginning. I would like to thank my co-authors, Adam J. Oliner for coming up with the idea of Carat in the first place, Matei Zaharia for building the Spark cluster computing system that I used for Carat data analysis, Petteri Nurmi for his help and expertise in machine learning algorithms since the beginning of my studies, Ella Peltonen for advertising the Carat project at every opportunity, and the University of California at Berkeley and the AMP Lab for hosting me during the development of Carat. I wish to thank the University of Helsinki, Department of Computer Science, where I did most of my work.

I would like to thank the FuNeSoMo exchange program and ICSI for arranging my research visit to Berkeley, CA, USA. My work on Carat would not have been possible without the funding from the Helsinki Doctoral Programme in Computer Science (Hecse). This funding enabled work on the project without diversions.

In Helsinki, November 17, 2014

Eemil Lagerspetz

Contents

List of Reprinted Publications

This thesis consists of an overview and four publications, referred to by PI-PIV. The authors have contributed to these publications as follows.

Context-Awareness with Data Gathering

Research Paper I: Joonas Kukkonen, Eemil Lagerspetz, Petteri Nurmi, and Mikael Andersson, "BeTelGeuse: A Platform for Gathering and Processing Situational Data," In *IEEE Pervasive Computing*, IEEE, 2009, vol. 8, no. 2, pp. 49-56

Contribution: The BeTelGeuse project was started by Petteri Nurmi. The author co-developed the system with J. Kukkonen. The author did roughly half of the design, implementation, testing, and analysis of the work. The author was the lead author of the publication with P. Nurmi and J. Kukkonen.

Mobile Operating System Energy Efficiency APIs

Research Paper II: Sasu Tarkoma and Eemil Lagerspetz, "Arching over the Mobile Computing Chasm: Platforms and Runtimes," In *Computer*, IEEE, 2011, vol.44, no.4, pp.22-28

Contribution: This publication is a survey of mobile platforms. The author was the lead author with S. Tarkoma. Particularly, the author gathered and wrote information on Java ME, Android, and Windows Phone 7, with special interest on network and energy monitoring.

Collaborative Energy Awareness

Research Paper III: Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma, "Carat: collaborative energy diagnosis for mobile devices," In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys '13)*, ACM, 2013, Article 10, 14 pages

Contribution: A. J. Oliner started the Carat project with the idea to collect battery information from iPhone devices. The author co-authored Carat with Oliner, and came up with the a priori slice weighting for inaccurate iOS battery measurements, as well as the current version of the Carat algorithm, Algorithm 1. The author did roughly half of the design, implementation, testing, and analysis of the work.

Research Paper IV: Kumaripaba Athukorala, Antti Jylhä, Eemil Lagerspetz, Maria von Kügelgen, Adam J. Oliner, Sasu Tarkoma, Giulio Jacucci, "How Carat Affects User Behavior: Implications for Mobile Battery Awareness Applications," In *Proceedings of CHI Conference on Human Factors in Computing Systems (CHI'14)*, ACM, 2014, pp. 1029-1038

Contribution: This work was directed by K. Athukorala. This publication used data gathered by the Carat Android application, and questionnaire responses from its users. M. von Kügelgen designed the questionnaire together with K. Athukorala. The author designed, implemented, and tested the questionnaire module for Carat that was used to gather questionnaire results from Carat users. The author created the method to match Carat user data with questionnaire responses, and analyze how often users stopped using Hogs and Bugs. The questionnaire responses were analyzed by Athukorala and von Kügelgen.

Chapter 1

Introduction

Smartphones are an integral part of our daily lives in many parts of the world. The value of a smartphone is in the connectivity and the applications it provides. These drain energy from the smartphone's battery, setting the time that the device will remain usable, after which the battery will need to be recharged. The development of exponentially more powerful smartphone hardware [1],[2, pages 8-11] and more complex applications has increased the energy demand, but battery technology has developed much more slowly. Watching movies on a smartphone can drain the battery in 3-6 hours, requiring a recharge during the day [3]. All of this means that energy is at a premium on the smartphone. It needs to be spent wisely and conserved where possible.

1.1 Motivation

Previous work has discussed the improvement of battery technology by the use of better materials [4, 5] but these technologies are not yet commercially viable for smartphone use. Modeling of battery characteristics has been a popular research topic. This includes modeling the state of charge (SOC) [6, 7], taking into account the temperature of the battery and battery deterioration over time [8], and the environmental energy impact of battery chargers [9].

Some work has taken the models online into mobile applications that are able to estimate aggregate energy use while the device is being used [10, 11]. Mobile power measurement has been studied further, attributing energy use to the screen backlight and communications [12], individual TCP/UDP packets and network events [13], security and cryptography in communications [14, 15], and operating system functions [16]. Application and

system-level power monitoring solutions for mobile devices in a laboratory environment have also been researched [17, 18, 19]. In addition, previous work has considered the energy consumption of video streaming [3, 20] as well as energy efficiency techniques for particular scenarios, such as communications, communication offloading, and gesture recognition [21, 22, 23, 24].

There is a rich body of research on mobile applications that optimize energy use from the user's point of view. A few improve the energy efficiency of the system [25, 26] while others consider user behavior and likely scenarios for optimization [27, 28, 29, 30].

Context awareness has been researched extensively [31] also in the context of energy efficiency [32, 33, 34]. Finally, there is some work on mobile energy diagnosis in the past [35, 16, 36].

However, there is no previous work on combining rich context information from multiple devices with energy awareness. Without this bridge, it is not possible to determine whether energy used by a device is normal for that workload and type of device or that device class. For example, the energy use of two identical smartphones, running the same applications, can be radically different, when they are running slightly different operating system versions, or connected to two different backup services.

1.2 Problem Statement

Previous research is unable to detect whether the measured energy consumption is normal for a given application or device. Indeed, it is difficult to ascertain what is normal for an application, without using several devices of the same type, running the same application version. Other factors than the device model also affect energy use. Battery temperature, background processes, Wi-Fi and cellular signal strength may change the energy use radically. We have gathered data from devices across the world, and cannot have physical access to the devices, or the source code of the applications that they run.

The goal of this thesis is to gather energy measurement data from hundreds of thousands of devices, running many applications, in heterogeneous environments, and construct models of what is normal in each context for each application. It is then possible to find energy abnormalities in the wild. These abnormalities cannot be discovered in laboratory conditions due to the rich interaction of the smartphone and its operating environment, for example, the effect of users moving around cannot be determined in a single laboratory location.

The author views applications as black boxes; we cannot determine the

energy use of an application as correct or incorrect, if it always stays the same. However, the author can compare the same application between devices, platforms, and OS versions, and applications that serve the same purpose with each other.

Figure 1.1: The methodology of this thesis includes empirical measurement and statistical analysis.

The work in this thesis is called *collaborative mobile energy awareness*. Over time, a collaborative mobile energy-awareness application gives increasingly accurate battery life advice to its users based on data gathered from all participating mobile devices, such as smartphones and tablets. Employing a collaborative mobile energy-awareness application with thousands of users allows gathering a large amount of data in a short time, which would take

years for a research team working on their own. This thesis uses this large and diverse dataset to answer various research questions. The questions explored by this work are listed below, with the rationale why the questions are relevant.

RQ1. Can we gather rich context data regardless of the hardware platform?

RQ2. What are the best smartphone platforms for energy and context data gathering?

RQ3. Is the change in the smartphone battery API battery level a reliable estimate for energy use?

RQ4. Is it possible for a collaborative mobile energy-awareness application to detect an energy bug injected into the community?

RQ5. Can a collaborative mobile energy-awareness application discover applications with higher than normal energy use in the wild (Hogs using the terminology of PIII)?

RQ6. Can a collaborative mobile energy-awareness application discover energy Bugs in the wild? These are defined as applications with higher than normal energy use, that affect only a subset of devices.

RQ7. How much does a collaborative mobile energy-awareness application improve the battery life of its users?

RQ8. Do collaborative mobile energy-awareness application users gain a better understanding of their device's energy behavior over time?

If RQ1 is answered in the positive, this work can reach a wider audience of users. If the answer to RQ3 this would be negative, our only option would be a hardware device for energy measurement, attached to the smartphone. This would make large-scale energy measurement impossible. A positive answer to RQ4 validates that the methodology of Carat is sound. This is examined in Chapter 3. The answer to RQ5 should be positive to ensure that this work can be used to detect and filter out Hogs, i.e. applications that regularly use more energy than the average application. RQ6 can then be properly considered. RQ7 is used to measure Carat's usefulness to its users. The long-term goal of Carat as an energy-awareness application is to improve its users' understanding of the battery and which activities are likely to reduce the time between recharges of the smartphone battery. This is measured by RQ8.

1.3 Methodology

Figure 1.1 shows an overview of the research methodology used in this thesis. The first step is to discover how to gather rich context data on smartphone platforms. Empirical battery level and time data for a set of devices and applications is then gathered. The author compares the battery change data on reference phones with power measurements conducted with a hardware measurement device, to verify that battery level and time data can be used as an accurate enough estimate for energy use. Once verified, the empirical data is used to calculate energy profiles for all application and device pairs. The author applies statistical methods to detect anomalous energy use in these pairs, and can present it to the user as recommended actions, such as killing an application or restarting another. Finally, the author has recorded the history of applications running on a device, and surveyed users with K. Athukorala to gauge user behavior changes. This allows answering questions such as "How often do users kill Hogs/Bugs/other applications?" and "How often do users who know about a Hog still run it?".

The methodology in this thesis is statistical, and as such is subject to the concerns of selection bias and sample size. Section 3.3.2 alleviates some of these concerns.

To gather data from smartphones in the wild, we published the Carat application in the Apple App Store[1] and Google Play[2], and advertised it on Reddit[3] and technology blogs such as Tech Crunch[4]. The application quickly gained thousands of users. Carat measures the time it takes for the battery to drain 1%, or, in the case of iOS, 5%. It records which applications were being used during each 1% or 5% drop in battery level. When a Carat user opens the application, it sends data it has gathered so far to us. The size of the data is still growing. At the time of writing, our dataset had more than 725,000 distinct devices, of which 54% were iOS and 46% Android. This large dataset from many devices lets the author establish reliable expected energy use of applications, mitigating the effect of user behavior on application energy use.

Figure 1.2 shows an overview of the power profiling methodology used in this thesis. The author starts by crowdsourcing power measurements and application information from a large number of devices. These are used to

[1] `https://itunes.apple.com/us/app/carat/id504771500`, visited November 17, 2014

[2] `https://play.google.com/store/apps/details?id=edu.berkeley.cs.amplab.carat.android`, visited November 17, 2014

[3] `http://www.reddit.com/r/compsci/comments/q1xxg/improve_your_iphones_battery_life_help_a_fellow/`, visited November 17, 2014

[4] `http://techcrunch.com/2012/06/14/carat-battery/`, visited November 17, 2014

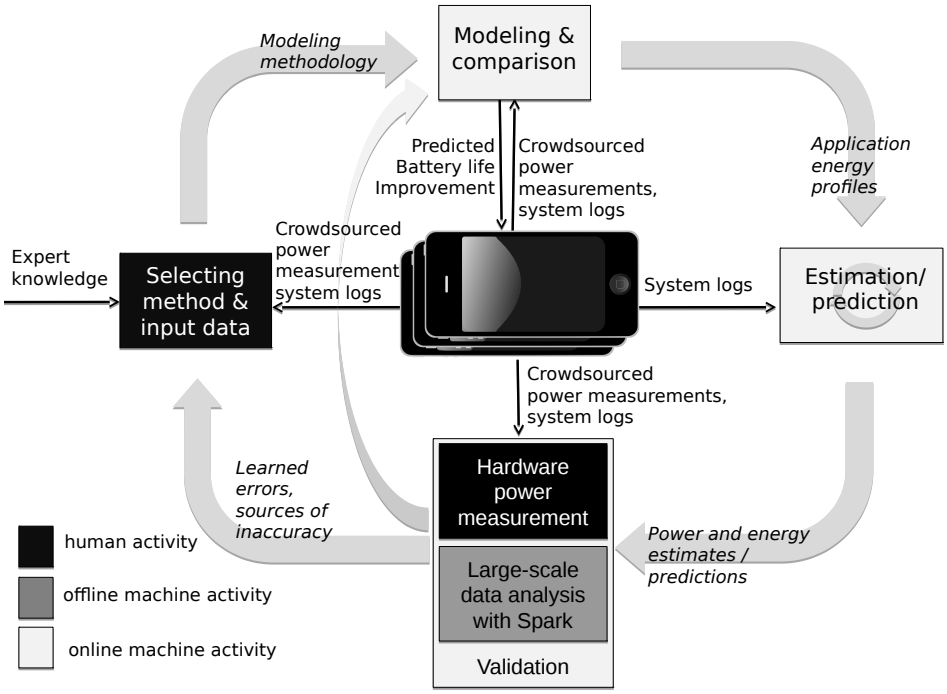


Figure 1.2: The control loop of Carat includes crowdsourced energy measurements and application activity logs, and its power profiles are validated with large-scale data analysis and hardware energy measurement. Figure adapted from Smartphone Energy Consumption: Modeling and Optimization [2]

construct a statistical power model. The model is validated or calibrated with a hardware power measurement device. Energy use of an application is then compared to energy use without it. These comparisons are used to predict the battery life improvement gained when the user stops using an application. The predictions are sent back to the users as they open the application. Predictions given by Carat are validated with hardware power measurements, and ongoing large-scale data analysis with Spark. Finally, the results are used to refine the power model for all users and applications.

1.4 Thesis Contributions

To the best of our knowledge, Carat is the first collaborative mobile energy awareness system, learning what is normal with help from its users and

environment [37]. Carat then detects anomalies from that normal behavior, and presents corrective actions to the user. For each abnormality, one can formulate corrective actions and calculate their expected energy benefits. In PI, we have shown that rich background context information can be made available to even older smartphones. This work answers RQ1. Such information can be used to improve mobile operating system energy awareness, which is still lacking. There is a need for a common API across mobile platforms, that unifies network connectivity, energy awareness, and user experience. Without such an API, it is difficult to diagnose the causes of battery drain on the mobile platform. PII motivates the smartphone platforms chosen for this thesis and answers RQ2. We have shown that battery level and running application information from a large number of users can be used to find and diagnose energy bugs (PIII). With better context information, the diagnosis can pinpoint the cause more precisely, as seen in Figure 16 in PIII. Finally, a questionnaire and data analysis of Carat users has revealed that collaborative mobile energy-awareness systems can improve the battery awareness of their users and improve their experience with smartphones (PIV).

Figure 1.3 shows the contributions in this thesis and their relationships with previous work. This work combines understanding of mobile operating systems and middleware, context awareness, energy awareness, and user behavior. The Figure shows two pieces of related work in the area of Bug Diagnosis. These are eDoctor [38] and MobiBug [39]. These systems diagnose traditional programming errors, not energy bugs. In the area of context awareness, and operating systems, we mention ContextPhone [40], an early context data gathering system aimed at researchers. This system can be seen as a precursor to BeTelGeuse, included in this thesis as PI. The remaining work in this category is included as PII.

The work in PIII, Carat, is the latest in a long line of mobile energy measurement and profiling systems, such as PowerScope [41], CasCap [34], eProf [16], and mPower [42]. These systems measure the energy used by the smartphone and its applications using various techniques, some have cloud support [34, 42]. These systems are not collaborative.

The purpose of Carat is to compare a large body of applications with each other, and application running on different devices, and find anomalous energy use. Carat considers applications as black boxes, and can compare the energy use of devices running the same application to determine if the application consumes an abnormal amount of energy on some devices. It can also compare applications with each other, finding applications that users should avoid, or energy-conserving alternative applications.

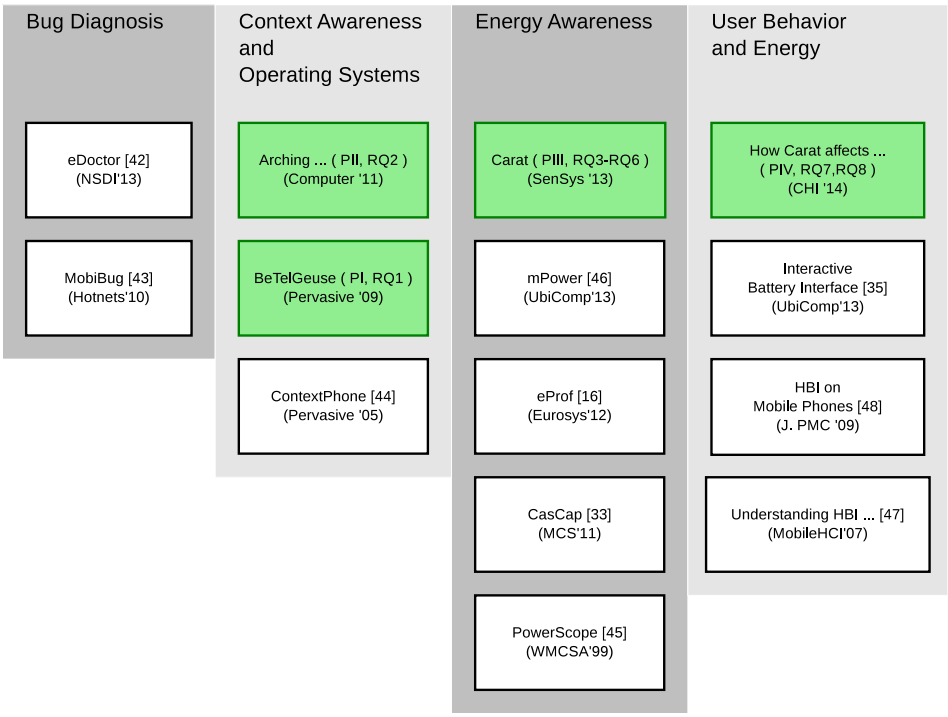


Figure 1.3: This work combines context awareness, mobile energy awareness, and mobile user behavior.

Carat cannot determine the exact amount of joules used by an application or a system configuration. A hardware power measurement tool, such as BattOr [43] or the Monsoon Power Monitor[5] is better suited for fine-grained absolute energy measurement. This work answers RQ3–RQ6.

Finally, the user behavior study on Carat users follows previous work in the area of studying user behavior from the energy viewpoint. This is also known as Human-Battery Interaction (HBI). Previous work [44, 45, 36] examines how willing people are to change their behavior to save battery life, how and when they charge their smartphone batteries, and how the visualization of remaining battery life affects the charging behavior. The study on Carat users is included as PIV. This work examines how a smartphone battery awareness application changes the behavior of its users over time, and answers RQ7–RQ8.

[5] `https://www.msoon.com/LabEquipment/PowerMonitor/`, visited November 17, 2014

1.5 Thesis Structure

This thesis consists of the original publications PI–PIV and the present introduction. Chapters 1 - 5 of this thesis aim to introduce the topic of collaborative mobile energy awareness, and explain our research questions and methodology in more detail. The introduction is structured as follows.

Chapter 2 briefly reviews the state of the art in mobile energy efficiency, including automatic operating system-level solutions, energy awareness systems aimed at users, and the use of context information. Chapter 3 introduces our work in collaborative mobile energy awareness. The chapter is divided into four sections. First, Section 3.1 discusses ways to improve system knowledge of the user's situation. Section 3.2 takes a look at the mobile operating systems available today, and Sections 3.3–3.3.5 introduce collaborative mobile energy awareness with the Carat system, the methodology used, and the current deployment. Finally, Section 3.4 discusses energy awareness with a user-centric viewpoint. Chapter 4 It discusses the impact of collaborative mobile energy-awareness applications, and outlines future work. Chapter 5 concludes the thesis.

Chapter 2

Mobile Energy Efficiency: State of the Art

This chapter discusses the domain of mobile energy efficiency and how it has been improved to date. Hardware-level energy efficiency solutions and device component improvements are out of the scope of this thesis, and will only be briefly mentioned. This thesis will also consider the problem of large datasets, which needs to be solved before a system for a large number of mobile devices can be deployed.

The work in mobile energy efficiency can be categorized by abstraction level into hardware, operating system, middleware, application, and collaborative solutions. They can also be categorized by level of interaction, into fully automatic, semi-automatic, manual, and awareness-only systems. Automatic systems work without input from the user or above abstraction layers. Semi-automatic solutions need some input parameters from the user or layers above. Manual solutions need the user or a component on a higher layer to turn them on and off as needed. Awareness systems, also known as informational systems or awareness-only systems, only provide information about what is using energy, and may give suggestions on how to improve the situation. Any improvement actions need to be carried out separately by the user or a software component.

Solutions in the operating system layer need deep knowledge of hardware and device state, middleware solutions take control of operating system components, like communications or storage for energy efficiency, and application solutions have a holistic view of the device and other applications running on it. Application solutions can also be used directly by the user. Carat is an application-level, *collaborative* energy-*awareness* system. Collaborative solutions require inputs from more than one device, and as such go beyond the application layer of a single device.

2.1 Middleware and Operating System Layer

Operating system and device component power efficiency and monitoring solutions in a laboratory environment have been researched in the past [3, 22, 23, 17, 24, 18, 19]. This research targets components of the smartphone with specific, mostly automatic improvements in mind. We will highlight some of these solutions on the operating system and middleware layers below. A deeper examination is available in Energy Management Techniques in Modern Mobile Handsets [25].

The Wi-Fi PSM or Power-Saving Mode automatically sets the Wi-Fi radio to a low-power state when there is little traffic. It is therefore an automatic, hardware-level solution. DVFS or Dynamic Voltage and Frequency Scaling is a solution to reduce power by reducing the input voltage and clock frequency of the CPU of a mobile device, when processing power is not needed. This needs to know how saturated the CPU is to work properly, so it is a semi-automatic solution.

At the operating system level, the Linux cpufreq governor uses DVFS to save power when the system is not heavily loaded. There are various policies that can be used to dictate how aggressively cpufreq saves power. Similarly, on many Android phones, when the battery level is low, the system activates power saving measures, such as screen brightness reduction and DVFS. This can also be turned on by the user on some phones. The energy benefits of DVFS have been considered in the past [46]. For our purposes, DVFS and other hardware- or model- specific techniques are not relevant, since they cannot be easily deployed on hundreds of thousands of devices.

For awareness on the hardware and operating system-level, the Android battery API gives information on how the battery is being drained and at what voltage. This comes directly from the smart battery of the device. A close relative of this is the Android power profile, which gives estimates of energy use of various features of the phone. This is based on tests by the device manufacturer, and may not be accurate in all conditions.

A well-known transmission efficiency technique on the operating system layer is Nagle's algorithm. The idea of this algorithm is to avoid small transmissions, improving the content-to-header ratio. This also improves energy efficiency, as it makes transmissions happen less often. This algorithm is used automatically.

The WakeLock system on Android allows the device to go to sleep when no processes are holding a WakeLock. On the other hand, this requires developers to take a WakeLock for longer processing tasks, and properly release the lock when the processing finishes.

On the middleware level, the benefits of offloading computation to more

powerful devices from the smartphone have been considered in the past [47, 48]. Many operation offloading systems have been developed over the years to improve the energy efficiency of the mobile device [49, 50, 51, 52, 53]. Most of these require the programmer to manually mark sections of the program for offloaded execution when possible. The offloading may then happen automatically as the program is run and execution speed of the mobile device and the bandwidth and latency of the network are determined.

2.2 Single Device Systems

Mobile power measurement has been studied in detail [12, 13, 9, 14, 15, 16]. Previous work has also examined the energy use of the mobile device on the application layer [59, 60, 61, 62]. This is necessary in order to improve application power efficiency [63] or predict it [64]. However, previous work has mostly focused on tracking the energy use of a single device or its applications. Table 2.1 lists some previous work in smartphone energy measurement and profiling in chronological order. Here many systems are self-constructive, which means that the power model is calibrated automatically. Laboratory power measurement is not required. Carat also belongs to this category, although hardware power measurements were made to validate its methodology.

There has also been work on improving the efficiency of network connectivity and location services [65, 66]. The battery impact of displaying advertisements within smartphone applications and websites has also been considered [67]. All of the above fall into the software energy efficiency systems category. Most of this work targets better energy awareness, while some introduce manual or semi-automatic energy efficiency improvements. Systems such as eProf [16] improve energy awareness by giving reports on the energy use of applications. Most Android energy saving applications, that target the user, offer automatic improvements. These include, e.g., JuiceDefender[1]. Some systems have cloud or server support [34, 39, 42]. This enables making decisions that take into account the entire system, not just a single user. In the case of our work, this is taken further by using the measurements of each device to make more accurate diagnoses on all devices.

[1] `http://www.juicedefender.com/`, visited November 17, 2014

Name/Authors	Year	Purpose
PowerScope [41]	1999	Energy profiling of device and processes
Joule Watcher[54]	2000	Fine-grained thread-level profiling
Nokia Energy Profiler [12]	2006-2007	On-device standalone profiler
Shye et al. [27]	2009	Energy profiling of device and components with a logger application
PowerTutor [11]	2009	Hybrid profiler based on PowerBooter
PowerBooter [11]	2009-2010	Short-term power model for components
BattOr [43]	2011	Portable power monitor
Sesame [55]	2011	Self-constructive on-device power model for device and components
PowerProf [56]	2011	Self-constructive API-level power profiler
MobiBug [39]	2011	Automatic diagnosis of application crashes
Carat [37]	2012-2013	Application energy profiling and debugging
eProf [16]	2012	Fine-grained power model for device, components and applications
DevScope [57]	2012	Self-constructive power model for device and components
AppScope [58]	2012	Fine-grained energy profiler for applications based on DevScope
eDoctor [38]	2012	Automatic diagnosis of battery drain problems
V-Edge [10]	2013	Self-constructive power model for device and components

Table 2.1: Previous work on smartphone energy measurement, listed chronologically.

2.3 Statistical Techniques

Our approach in the Carat project is a form of statistical debugging, where anomalies are called Bugs [68]. Such methods have been used to identify code paths correlated with failure [69, 70], concurrency bugs [71], shared influence, i.e., surprising behavior that is correlated in time [72, 73], invariant violation [74], and configuration errors [75]. In the field of security, anomaly-based intrusion detection has a long history [76, 77, 78]. Recently, statistical methods were used to diagnose energy problems by comparing the behavior of an application at different times on a single device [35]; this kind of approach cannot distinguish whether the energy behavior of the application is normal or not. It can only determine how the energy use changes over time.

2.4 Collaborative Systems

Distributed diagnosis of Windows application errors has been researched in 2009 [79]. The system did not consider energy aspects, and was not collaborative. Also, CarrierIQ[2] collects detailed measurements by integrating with the mobile platform. The system is not collaborative, but collects measurements from a large number of devices to a central location.

Systems like SETI@Home [80] and the Application Communities project [81] use the community to distribute work. Community-based security research includes finding the root causes of problems [75] and shared intrusion prevention of known exploits [82, 83].

Collaborative mobile application debugging has been suggested in the past [39]. This system was designed to collect traces of smartphone behavior in a centrally controlled manner, requesting only subsets of the device community to run tests in order to isolate the problem. This system was aimed at application developers, and did not consider the energy use of applications. To the best of our knowledge, the system was not publicly deployed. Device Analyzer from the University of Cambridge [84] also collected rich context information from a large number of Android smartphones. This was a successful exercise in large-scale data collection. The system did not analyze application energy behavior, but gathered information that could be used for energy analysis. In earlier work, Verkasalo and Hämmäinen also gathered data on over 500 mobile handsets [85]. There is also prior work for generating alerts based on statistical indicators of the

[2] `http://www.carrieriq.com/`, visited November 17, 2014

entire community, in the area of file systems [86] and peer-to-peer networks [87].

To the best of our knowledge, Carat is the first collaborative mobile energy-awareness system. A device participating in Carat automatically improves recommendations for all devices that share applications with it.

2.5 Large Datasets and Scalability

There is prior work on handling large datasets [88, 89]. The phrase *Big Data* is used to refer to data from a large number of sources, that is too diverse to handle with traditional methods, or too large for a given data processing system to handle efficiently. Ji et al. [88] define Big Data as difficult to handle with traditional data processing tools, such as databases. Processing such large datasets requires a redesign of the algorithm with parallel processing efficiency in mind. Particularly when processing a large dataset distributed across many computers and CPUs, it is not possible to maintain a frequently updated global state of the algorithm. This is one reason why solving a Big Data problem with a well-known algorithm, can be difficult. The algorithm may require fundamental changes, such as splitting the algorithm state among the computing machines, and merging it in the end. Some algorithms cannot be easily converted to this form. For example, in a loop-based algorithm, strong dependencies of the current iteration on the previous one create a situation where iterations cannot be run in parallel, but must be run in sequence, limiting the algorithm to a single computer. In this kind of case, a replacement algorithm may need to be devised, or the problem to be solved redefined in a way that better fits distributed data processing.

2.5.1 Centralized Programming, Distributed Execution

In 2012, Twitter integrated machine learning into their large-scale distributed computing system based on Hadoop [90]. Their work shows some of the issues and solutions required in using machine learning for large datasets. Large datasets from many sources are highly valuable to companies and large organizations [91]. Information such as customer behavior and customer categorization can help target product development better to the customers.

The computing paradigms for managing large datasets are different from traditional approaches [92, 88]. Currently, most large dataset processing is done using a *centralized programming, distributed execution* type of system, such as Apache Hadoop, MapReduce [89], or Spark [93] (also published

earlier as a technical report [94]). These typically run on a set of either physical or virtualized hardware, called a cluster. They take over the entire cluster and distribute tasks of one job to the cluster. The fraction of nodes or cores per node used by these systems can be configured; another option is to use a system like Apache Mesos [95, 96]. Mesos is a resource management system that can be used to run various data processing frameworks in parallel, such as Hadoop and Spark.

The main advantage of a centralized programming model with distributed execution by the system is that for each new program, the distribution of tasks does not need to be reprogrammed. The programmer creates a single program, and Hadoop or Spark will distribute any tasks in it that operate on top-level collections of data items. This has a number of benefits.

First, the programmer has no access to the entire collection, only the part visible to a single task. There can be no indexing errors on the top level collection. Second, computation nodes will always handle their part, and their part only. Results will not be duplicated. Third, retries can be handled by the framework. When a task fails, the system knows which tasks to run again on which pieces of data. Fourth, The key-value model used by Hadoop and Spark allows grouping of data by any attribute. This makes it easy to focus tasks on details while being able to leverage the entire cluster.

2.5.2 Clusters and Cloud Computing

The basic requirement for a distributed computing system such as Spark is a cluster of computing machines. These may or may not be heterogeneous in terms of processing power and memory capacity. A cluster may consist of physical hardware, or virtual machines (VMs). The below lists some of the benefits of VMs.

First, a VM-based architecture is easily scalable. One can create a bundle of the computing software and choose the hardware specifications, and then spin up any number of VMs with those specifications. Second, there is no up-front cost of buying hardware. Once there is a prototype, VMs running it can be started, and terminated when done, paying only for the time they were used. Third, as data storage needs grow, storage space can be increased transparently of the software architecture. The software can keep running and writing to storage without interruptions.

Using a cluster of VMs for data processing is called *cloud computing*. Using VMs and cloud computing has its requirements and challenges. Clouds have to manage large computing facilities and multiple simultaneous requests and operations similarly to grid computing [97]. Because of the ephemeral nature of VMs, resources can seem infinite in the cloud [98], but there are

very real limits that can be enforced in various ways. Foster et al. [97] and Armbrust et al.[99] discuss the role of the cloud in distributed computing in more detail.

2.5.3 The MapReduce Paradigm and Spark

MapReduce is a popular distributed computing paradigm developed by Google researchers Jeffrey Dean and Sanjay Ghemawat [89, 100, 101]. In MapReduce, the programmer writes two functions, *map* and *reduce*. Data items are presented as key and value pairs. The map function represents a transformation from a data item to another, and only that single data item is available in the programming context. The reduce function takes two data items with the same key and produces a data item of the same type. When called successively on the entire data set, it effectively reduces the set to one value per key. A MapReduce program to do a word count of the entire Wikipedia could look like this:

$$map(word) \rightarrow (word, 1)$$

$$reduce(word, count, count2) \rightarrow (word, count + count2)$$

It is easy to see that the map and reduce functions are very simple, when compared with a full distributed program that accomplishes the same task. It is also easy for the MapReduce system to execute these functions in parallel, since they only depend on the input value. The map and reduce functions are then stored on a *master* machine, that schedules them to be run on *workers*.

Several open source MapReduce implementations have been developed. Hadoop [102] is one of the most popular. Hadoop has since been rewritten as YARN [103].

Zaharia et al. have presented the idea of *Resilient Distributed Datasets* (RDD) in their paper [93] published in April 2012. Spark [104] is an open source implementation of the RDDs.

An RDD is a collection of data items. The RDD is *partitioned* to worker machines, similarly to tasks in MapReduce. The RDD is *read-only*, and modifying it will create a new RDD. The RDD is a *lazy* structure, as is common in Scala [105], the language Spark is written in. Spark has three advantages when compared with the MapReduce paradigm.

First, the RDD remembers the sequence of operations that produced it in the form of a *Lineage*. This means that only the operations and the latest result set need to be stored. Second, *Persistence*, or *caching* allows a user to moderate the storage strategy RDD uses, e.g., in-memory only

Transform.	Data operation	Meaning
$map(func)$	$RDD[V] \rightarrow RDD[W]$	MapReduce-like map, uses a function $func$ for every item in the data set of type V and returns a new set of type W
$flatMap(func)$	$RDD[V] \rightarrow RDD[W]$	Similar to map, but every input item can produce zero or more output items that will be combined into a flat sequence
$filter(func)$	$RDD[V] \rightarrow RDD[V]$	Result RDD will contain only the items for which the Boolean function $func$ returns true
$groupByKey()$	$RDD[(K,V)] \rightarrow RDD[(K, Seq[V])]$	Collects all the data sets related to each key and returns them as a key and sequence of the corresponding data items
$reduceByKey()$	$RDD[(K,V)] \rightarrow RDD[(K,V)]$	Reduces or aggregates the data items related to each key like MapReduce's reduce

Table 2.2: Some of the main Spark RDD transformations, which are performed lazily. The whole API document is available on [104].

or the memory and the disk. This functionality makes computing faster, when the data is cached in memory. Caching is a fault-tolerant feature, meaning that lost cached partitions can be recovered via the lineage. Third, *Data locality*, or *partitioning* allows the user to control the number of data partitions, and the placement of data via the partitioner interface.

Together these features make RDD/Spark more effective than a basic MapReduce implementation, as Zaharia et al. have shown in their article [93]. However, this comparison was made against Hadoop, not YARN. The performance of YARN may be improved as it contains some of the improvements in scheduling that Spark also uses.

The programming model of Spark [93] is a superset of that of Hadoop. There are a number of operations in Spark that are not available in the MapReduce model. Spark operations are categorized into two groups: tranformations, shown in Table 2.2, that operate on data items to produce other data items, and that are lazy; and Actions, shown in Table 2.3, that are similar to reduce, cause evaluation of the transformation chain leading to them, and produce result values. The second column of both tables describes

Action	Data operation	Meaning
$reduce(func)$	$RDD[V] \rightarrow V$	MapReduce like reduce, but for the entire dataset. Uses a function $func$ to aggregate the data items
$foreach(func)$	$RDD[V] \rightarrow Unit$	Does the same operation $func$ to each data item, does not return anything
$count()$	$RDD[V] \rightarrow Long$	Returns a count of the data items in the RDD
$collect()$	$RDD[V]$ $\rightarrow Array[V]$	Returns the data items to the master as an array of elements type V. This should only be done to small enough collections.
$first()$	$RDD[V] \rightarrow V$	Returns a first item of the RDD, same as the first element of $take(1)$
$take(n)$	$RDD[V]$ $\rightarrow Array[V]$	Returns n first items of the RDD as an array
$broadcast(obj)$	$obj \rightarrow$ $Broadcast[obj]$	Makes the current version of the object available for all the nodes in any function context
$saveAsText$- $File(path)$		Saves the RDD to the given file system path (local or distributed) as text files
$saveAsObject$- $File(path)$		As $saveAsTextFile$, but writes serialized object files that are easy to read again to Spark

Table 2.3: Some of the main Spark RDD actions, which are performed immediately as opposed to the Spark transformations presented in Table 2.2.

the change in data types, if any, that results from each transformation or action. For example, a *map(func)* operation on a dataset of elements of type V will result in a new dataset of type W, where W is the return type of $func$. The entire API is documented on the Spark website [104].

2.6 User Behavior

Human interface studies have shown that 80% of mobile users will take action to improve their battery life [44]. The way the battery level is displayed to the user can have a large effect on how often users charge their devices [45]. There is a rich body of research on mobile applications optimizing energy use from the user's point of view [25, 26, 27, 28, 29, 30]. This work is mostly on awareness systems that examine smartphone activities and user behavior. We have found that an energy-awareness application can change user behavior for the better [106].

Mobile energy diagnosis has been studied in the past [35, 16, 36]. These works target debugging of energy issues on smartphones. Some of the work is able to detect anomalous behavior when compared to past measurements. These systems are not collaborative. The changes in user behavior when given feedback they can act on has been studied in household energy awareness [107], but we are not aware of any user behavior studies of mobile energy-awareness applications.

2.7 Context and Energy Awareness

The definition of context awareness, and what context is, has been discussed in depth in mobile computing research [108, 109]. Context awareness and how to use it has been researched extensively [32, 33, 31, 34, 40]. Systems such as ContextPhone [40] and BeTelGeuse [110] were developed to gather rich context information for other applications. In our work, we combine context available on the mobile device (running applications) with energy use, and show to the user the applications whose energy use is abnormal in the community. There are two levels of context in our work: the local context on the mobile device that is running applications, and the global context of the community, and how that application behaves in the community at large. By comparing these two, we can determine if the energy use of a given application is normal.

Chapter 3

Collaborative Mobile Energy Awareness

This chapter discusses collaborative mobile energy awareness with Carat and related components. Collaborative mobile energy awareness builds on context and battery data aggregated from many mobile devices. Collaborative mobile energy-awareness applications construct a model of battery drain for devices in the community. The more data from more devices, the more accurate this model will be. Collaborative mobile energy awareness can therefore be split into the components of context data gathering, energy and battery data gathering from operating system APIs, the analysis of data from many devices, and the user experience.

To the best of our knowledge, Carat is the first collaborative mobile energy awareness system. A device participating in Carat automatically improves recommendations for all devices that share applications with it. This helps us quickly increase the accuracy of the recommendations that Carat gives its users.

3.1 Data Gathering for Context Awareness

To accurately attribute energy use to its cause, one needs as much context information about it as possible. To gather context information, we can rely on the API provided by the mobile device, or use a dedicated context data gathering tool such as BeTelGeuse [110]. This tool gathers data from the internal sensors of the smartphone as well as other sources, such as Bluetooth sensors and arbitrary Internet data sources. A data gathering framework such as this is essential for correct attribution of energy use.

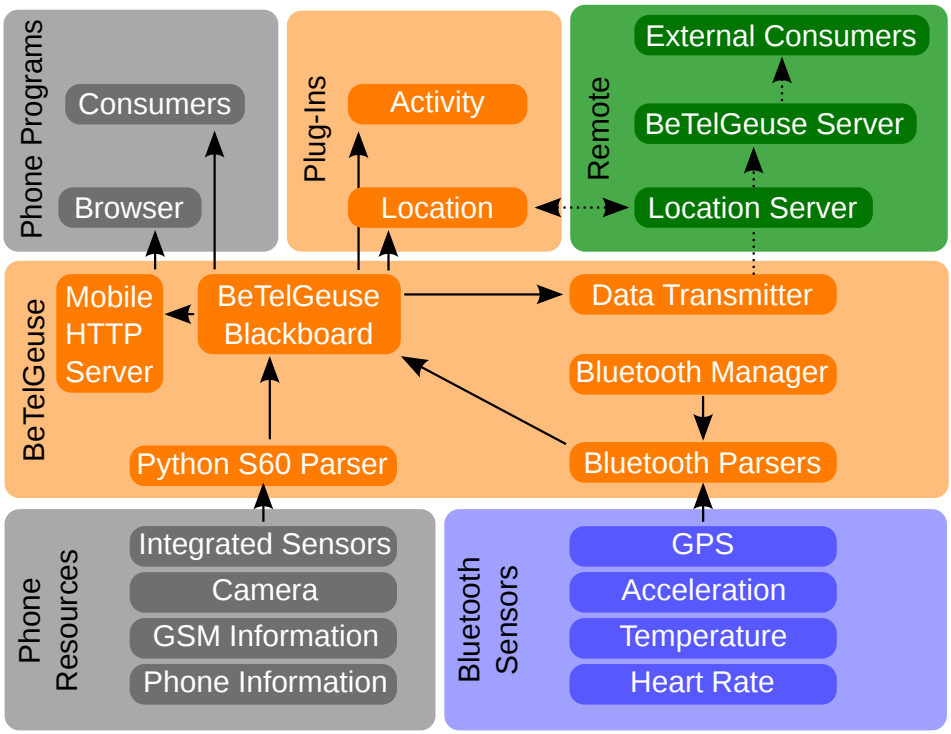


Figure 3.1: The core of BeTelGeuse is the Blackboard. The architecture of BeTelGeuse follows the microkernel principle.

PI gives an overview of the BeTelGeuse system. When the system was published, common smartphones did not have many internal sensors, and so Bluetooth was our main focus. The system was designed to be multi-platform, and answers RQ1. At the time of publication, the system was in active use for data gathering for location and context-awareness projects at the University of Helsinki. Figure 3.1 shows the architecture of BeTelGeuse. It consists of a minimal core with a blackboard for data access, and several plug-ins. Various sensor data parsers, as well as transmitting and saving context data are implemented as plug-ins. BeTelGeuse refines sensor measurements into more meaningful information, such as accelerometer data into user activity. This can be accomplished by a sensor parser or a separate plug-in that reads measurements from the blackboard and refines them. BeTelGeuse also allows data access via HTTP. BeTelGeuse can be seen as a precursor of the technologies present in current smartphones, such as the sensor hub of the Samsung Galaxy S4, and the HTML5 APIs for

mobile sensor data access. Sensor hubs are currently being researched also for the Internet of Things (IoT) [111]. There a plug-in architecture is highly useful.

3.2 Mobile Operating System Energy APIs

While context data for collaborative mobile energy awareness could be gathered using Bluetooth sensors or over a network connection from IoT devices, the current generation of smartphones does not use such devices most of the time. For collaborative mobile energy awareness to be truly effective, we need as large a user base as possible. Therefore we have opted to use only the sensors integrated in the smartphone.

Mobile operating systems such as Android and iOS provide APIs for determining the current battery level and the settings of the phone, as well as the list of applications currently running. We can use this information to implicate processes, settings, and communications hardware for the energy that is used by the mobile device.

PII contains a survey of current smartphone platforms [112], and their energy features. The energy data provided by each platform is different. Some platforms lack features, and others provide coarse information only. For example, the battery level can be obtained at 1% granularity from an Android device, but an iOS device will only give it at 5% granularity. Similarly, the list of running applications can be obtained on iOS and Android, with different semantics for what is considered "running", and cannot be accessed at all on Windows Phone 8 devices.

This work has been useful in determining how to best gather smartphone energy data, and which mobile operating systems to target. On most mobile platforms, we can obtain the list of running processes and the battery level. On Windows Phone 7, 8 and up, the process list cannot be obtained by a regular application. This is why we chose to target iOS and Android. This answers RQ2.

3.3 Collaborative Mobile Energy Awareness

To get a detailed picture of the energy use on a smartphone, one needs to gather data on the battery level and active system settings and applications on the smartphone, over time. This can then used to implicate applications for the battery drain that occurred when they were active. However accurate we make that picture, it cannot determine whether the observed energy use of an application is normal or abnormal. To diagnose excessive energy

consumption of applications, or energy bugs [113, 114], we need data from multiple devices.

3.3.1 Carat Overview

The author developed Carat[1] in collaboration with UC Berkeley [37] and published it in the Apple App Store[2] and Google Play[3]. Carat quickly got hundreds of thousands of users, and the dataset is still growing. This work answers RQ3–RQ6, as well as RQ7 on the system level. Specifically, this work shows that using the battery level and time interval to measure energy use can reach precision near hardware energy measurement instruments, collaborative mobile energy-awareness applications can detect injected energy bugs as well as those in the wild, including those that affect a subset of devices, and Carat improves the battery life of its users by 10% to 41%.

3.3.2 Large-Scale Data Analytics

At the time of writing, the data in the Carat project had over 100 million samples, from over 725,000 devices from over 200 countries. 96 countries had more than 100 Carat users, and 47 countries had over a thousand. Figure 3.2 shows the distribution of Carat users. Countries colored darker blue have more users. The number of users is marked on the map for the top ten countries. As can be seen on the map, Carat is not limited to the developed world, or a limited set of countries. There should therefore be no cultural bias in the user base. Carat has been available in the Apple App Store and Google Play for more than two years now, which puts the application within reach of most smartphone users. It has been advertised in technology blogs and newspapers, both printed paper and online, mostly in the US and Finland. In PIV, we see that Carat is not biased towards the younger or more tech-savvy crowd.

At the time of writing, the size of the data in the Carat project was over 2 TB, or 2,000 GB. This is too large to fit into the memory of most computers. Dealing with datasets larger than the available computer memory is one of the challenges of Big Data. Systems such as Apache Hadoop based on the MapReduce paradigm [89] and Spark [94] have been developed to handle problems of this scale. The typical approach is to employ massively parallel algorithms, in a cluster of computers, so that each computer only computes

[1] `http://carat.cs.berkeley.edu/`, visited November 17, 2014

[2] `https://itunes.apple.com/us/app/carat/id504771500`, visited November 17, 2014

[3] `https://play.google.com/store/apps/details?id=edu.berkeley.cs.amplab.carat.android`, visited November 17, 2014

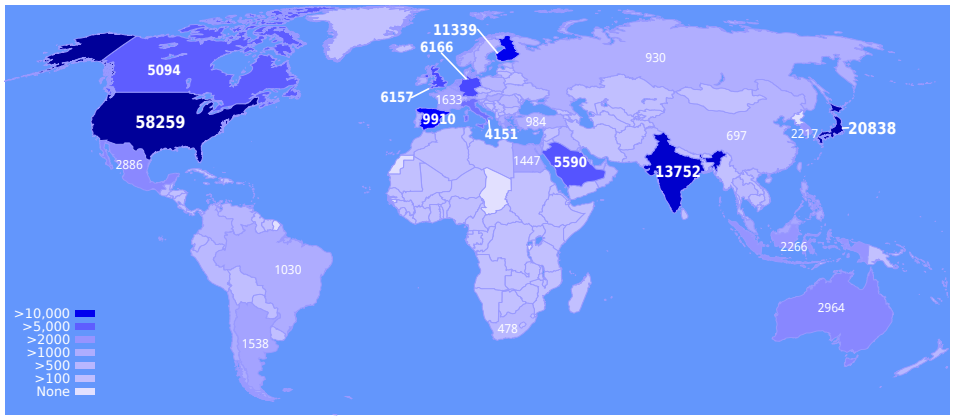


Figure 3.2: The Carat community is distributed across the entire world.

operations on a small slice of the data. Spark takes this further, keeping the data in the memory of the computer that read it, making iterative operations on the same data fast to compute.

In order to detect anomalous energy behavior in this dataset, the author needs to calculate the specification of what is normal. This depends on the application being used, and in case the application itself is the cause of the anomaly, on all of the data where the application is not being used. To be able to do this for the entire set of applications and users in the data, the author used the Spark cluster computing environment[4], running on a cluster of 10 virtual machines in Amazon's Elastic Compute Cloud (EC2). Each machine consisted of 8 virtual CPUs and 68 GB of memory. The machines shared a common filesystem, Apache HDFS, where all data was placed before computations. Results of the statistical analysis were stored in Amazon's storage service, S3.

3.3.3 Data Collection

In our work on Carat, context data is gathered using available mobile operating system APIs. The data gathered includes, but is not limited to:

- a statistically unique, randomly generated, time-based user id,
- a timestamp, in seconds since January 1^{st} 1970 00:00:00 GMT,
- the device model and OS version of the device running Carat,

[4] `http://spark.apache.org/`, visited November 17, 2014

- the list of running applications, their process/package names, process IDs, and other information,
- the memory use of the device, broken into free, used, active, and inactive, and
- the network status of the device, either disconnected, mobile, Wi-Fi, or WiMax.

This data is gathered on both the iOS and Android platforms of Carat. On Android, we gather more details about applications, their developers' digital signatures, and version history.

The data obtained from operating system energy APIs in Carat includes:

- the battery level, in percent,
- the timestamp when the battery level last changed,
- the current battery voltage, in Volts,
- the battery temperature, in degrees Celsius,
- the battery health, such as "Good" or "Overheat", according to thresholds set by the manufacturer,
- whether the battery is charging or discharging, and
- the charger type, one of none, AC, or USB.

The battery level has only 5% granularity on iOS, and 1% granularity on Android. In addition, Android's energy APIs allow reading the battery voltage, temperature, and health, while iOS does not. Future work on collaborative mobile energy awareness will investigate how to use the extended information obtained from Android devices to improve Carat's accuracy on iOS.

3.3.4 Method

The battery level change within a time interval, $r = \frac{b_1 - b_2}{t_2 - t_1}$, henceforth referred to as the *rate*, represents mean energy use within that time period. These rates are approximately normally distributed according to the Central Limit Theorem, as explained in PIII. To determine the expected energy use of an application a, we take all the rates with the application running in our dataset: $r_a = r_1, r_2, r_3, ..., r_n$. We then calculate the expected value (EV) of that distribution of rates. To get the expected energy use of all

applications, we take all the rates r_A from iOS or Android, depending on which platform's applications we are interested in, and calculate the EV. We then compare r_a with r_A using the difference d of their rate distributions' 95% confidence error bounds, $d = \mu_a - e_a - \mu_A + e_A$.

If the EV of r_a is greater than that of r_A, and the distance between their 95% confidence error bounds is positive, a uses significantly more energy than the average application. In this case a is a *Hog*. The benefit of not running a is then the difference between the EV of r_a and r_A, $\pm e_a + e_A$.

We do a similar comparison between r_a and all the rates of device d with a running, or r_{da}. If the same condition holds, d has a significantly higher energy use when running a than other devices, indicating a is an energy *Bug* on device d.

Finally, we rank the battery life of Carat users with the EV of their rate distribution. We take the user's percentile in the distribution of all users, as their rank. We call this rank the J-Score. An example user's J-Score is shown in Figure 3.4.

As we gather more data, the expected energy use and its error bounds change, and applications change their status between Bugs, Hogs, and regular applications.

Algorithm 1 outlines the process of calculating Hogs and Bugs in Carat, as well as the J-Score. The algorithm is given the set of all observed energy drain rates in Carat, and their frequency distribution *aDist*. We can then take all the rates for an application, and the rates without that application (but with all other applications), and determine the three main statistics used in Carat: the expected value, 95% confidence error, and number of samples. Based on comparing these, we can determine whether the application is a Hog or a Bug.

Figure 3.3 shows this comparison, where μ_1 is the expected value of the rate distribution for the application being considered, and μ_2 the expected value for rates without that application. The application is then a Hog if the difference d' between the error bars of the distribution is positive, and the application's battery drain rate's expected value is higher than that of other applications.

The process is similar for Bugs. The main difference is that we first take all rates from a given user id, and then from other user ids. We then consider an application that is not a Hog, for that user id and its complement. We compare the expected values for the distributions of *(id, app)* and *(¬ id, app)* similarly as with Hogs.

Finally, the algorithm records the expected value, 95% confidence error, and number of samples, for each user of Carat. The expected values are used

Algorithm 1 Detect Hogs, Bugs, and calculate J-Scores. This algorithm has not been published before. PIII contains an earlier version.

Precondition: *allRates* contains observed energy drain rates

```
1: function analyzeRates(allRates)

2:     Hog detection
3:     for app ∈ allApps do
4:         filt ← allRates.filter(app in _.allApps)
5:         filtNeq ← allRates.filter(app not in _.allApps)
6:         stats ← statistics(filt)
7:         statsNeg ← statistics(filtNeg)
8:         isHog,severity ← getDist(stats, statsNeg)
9:         if isHog and severity > 0 then
10:            store hog, stats and statsNeg
11:        end if
12:    end for

13:    Bug detection
14:    for id ∈ allIds do
15:        fid ← allRates.filter(_.id = id)
16:        notFid ← allRates.filter(_.id ≠ id)
17:        Consider apps reported by id, omit hogs
18:        fidNonHogs ← fid.map(_.allApps) \ Hogs
19:        for app ∈ fidNonHogs do
20:            appFid ← fid.filter(app in _.allApps)
21:            appNotFid ← notFid.filter(app in _.allApps)
22:            stats ← statistics(appFid)
23:            statsNeg ← statistics(appNotFid)
24:            isBug,severity ← getDist(stats, statsNeg)
25:            if isBug and severity > 0 then
26:                store Bug, stats and statsNeg
27:            end if
28:        end for
29:        scoreStats ← statistics(fid)
30:        Save scoreStats for J-Score calculation
31:    end for
32:    Write J-Scores based on saved scoreStats
33: end function
```

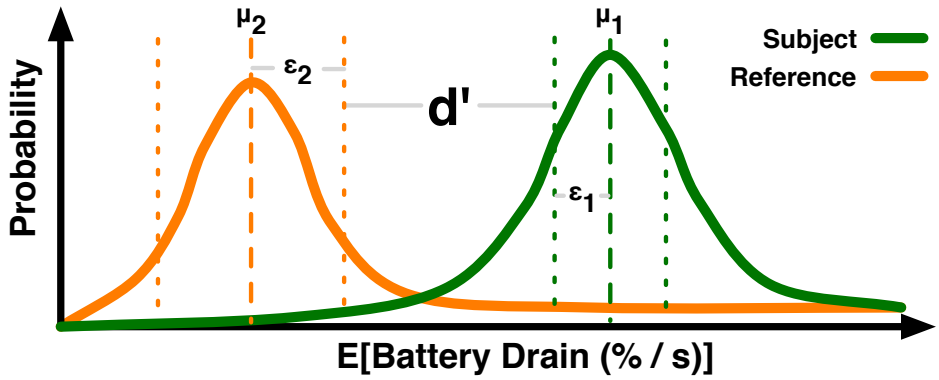


Figure 3.3: We define an anomaly as a configuration with a statistically significant, higher than average energy use.

to give users a ranking in the community, called the J-Score. An example user's J-Score is shown in Figure 3.4. The J-Score is simply the percentile of the user's energy drain rate's expected value in the distribution of all the expected values of all the users. Thus a J-Score of 64 for user u means that 64% of users have a higher expected battery drain rate than u. The battery drain rates are measured in %/s, meaning that a higher value results in a shorter battery lifetime. This makes u part of the top 36% of Carat users in battery life.

3.3.5 Implementation

Our work in the Carat project uses the Spark distributed computing environment described in Section 2.5.3. We run Spark on a cluster of VMs in the Amazon Elastic Compute Cloud (Amazon EC2) [115]. The system has also been run at the University of Helsinki on OpenStack [116].

Figure 3.5 shows an overview of Carat's architecture. Data is collected by Carat applications running on many iOS and Android smartphones. The data is sent to load-balanced Java-based servers that store it in a large, highly available storage service, such as Amazon S3. Every couple of days, an Amazon EC2 cluster of 10 large VMs is started. The Carat algorithm is deployed on it, and the analysis is run on the entire dataset. The algorithm generates J-Scores and Hog and Bug reports for all users that have sent us enough data for analysis. For best results, this means sending us at least a hundred samples, or running Carat for a week on iOS or a couple of days on Android.

Figure 3.4: The Carat application shows actionable recommendations to the user when opened. The second tab contains the J-Score and other information about the device.

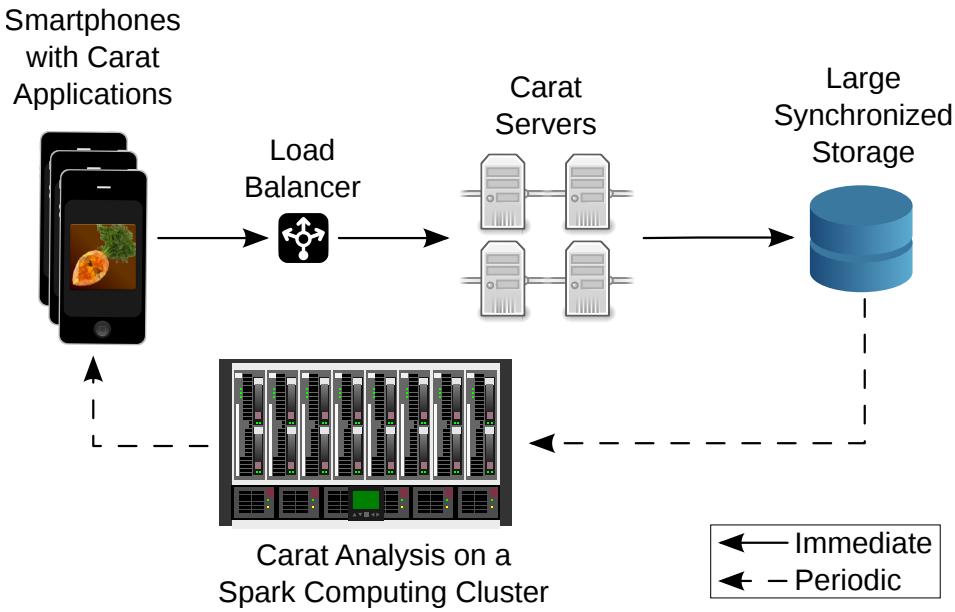


Figure 3.5: The architecture of Carat includes load-balanced servers for reception of data, and a computing cluster for periodic data processing.

The conditions μ_1 and μ_2 that Carat calculates error bars and differences for include:

1. an OS version and the latest OS version for the platform,
2. a device model and all other models ($\neg\mu_1$),
3. a running application and all other applications ($\neg\mu_1$), and
4. a running application on a particular device, and the same application on any other device.

The result from 1., if anomalous, can be seen in the Carat application when clicking the OS version on the My Device screen, shown in Figure 3.4. The second result is available by clicking the model name on the same screen. The results of 3. and 4. are shown in the Hogs and Bugs tabs of Carat, respectively.

3.3.6 Results

The algorithm shown in Section 3.3.4 is run for every user, application, and user-application combination every couple of days. On July 25, 2014, the

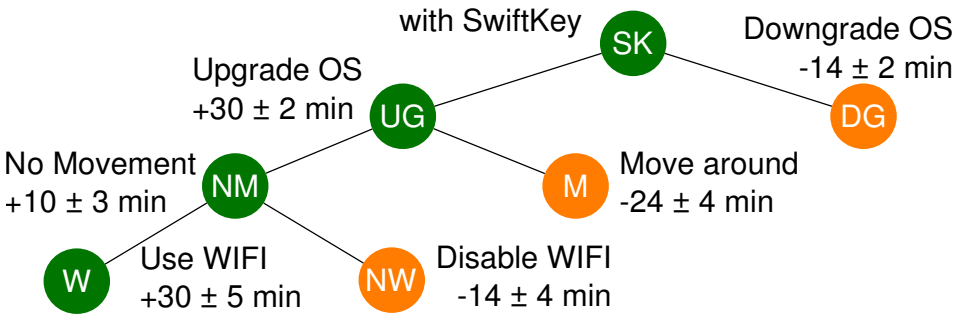


Figure 3.6: Carat can determine the battery life of a smartphone running an application under various conditions.

algorithm produced battery life estimates for 584,568 users that had sent enough data to us. 54% were iOS users, and 46% Android users. The algorithm found these users using 324,189 different applications, 4,613 different types of devices on 222 OS versions. Out of these applications, 87% behaved normally, but 13% (41,801) were Hogs or Bugs. The algorithm detected 26,885 Hogs, of which 9% were iOS Hogs and 91% Android Hogs. Android has more Hogs since the average energy use of an Android application is much lower than that of an iOS application, making the Hog threshold lower on Android. Of the users, 48% had at least one Bug. There were 14,916 Bugs (0.04% of all user-application combinations). Of the Bugs, 38% were on iOS, and 62% on Android.

The method of Carat is not limited to the conditions described above. In PIII, we describe the debugging of energy bugs found with our method. This process is shown in Figure 3.6 for one energy hog, the SwiftKey virtual keyboard on Android devices. This application uses more energy on older operating system versions, when the user is moving, and when Wi-Fi is disabled. This is typical for applications that require a network connection to function. SwiftKey was a hog, which means that the energy use for all of its users was higher than that of other users. The debugging tree in Figure 3.6 shows typical behavior for applications that require the network, which confirms that the application is a hog as a whole, and cannot be split further into normal and anomalous cases. We can compare this with the Kindle application, which suffered from a bug causing higher energy use when on 3G. The diagnosis tree for this is shown in Figure 3.7. We can see that on average, users using Kindle get longer battery life than other users, probably because most of the phone hardware is idle when reading e-books. Remember that Kindle was a Bug; its energy use is below average

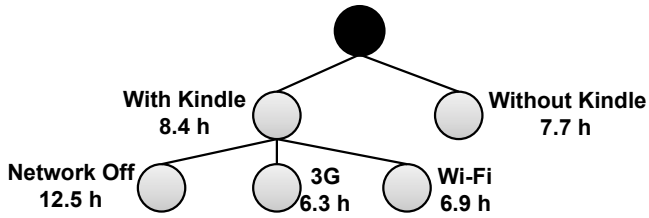


Figure 3.7: Kindle suffered from a bug that lowered the battery life of devices on 3G.

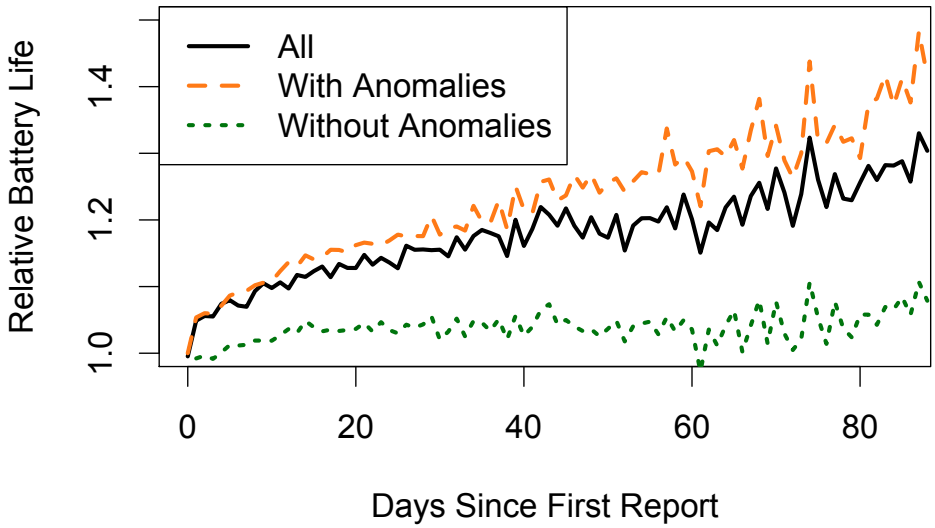


Figure 3.8: Carat increases the battery life of its users by 10 to 41%.

as a whole, but a group of users experienced anomalous energy drain when using it.

We can see that when 3G is on, battery life is reduced, even when compared with other users. The cause was incorrect handling of the connection on 3G when synchronizing book positions and nodes on WhisperSync, Amazon's synchronization service built into Kindle.

The above Hogs and Bugs are examples of many. Some more are listed in PIII. More current examples are shown on the Carat website[5]. This answers RQ5 and RQ6.

Finally, Carat improves the battery life of smartphone users with energy

[5] `http://carat.cs.helsinki.fi/statistics/`, visited November 17, 2014

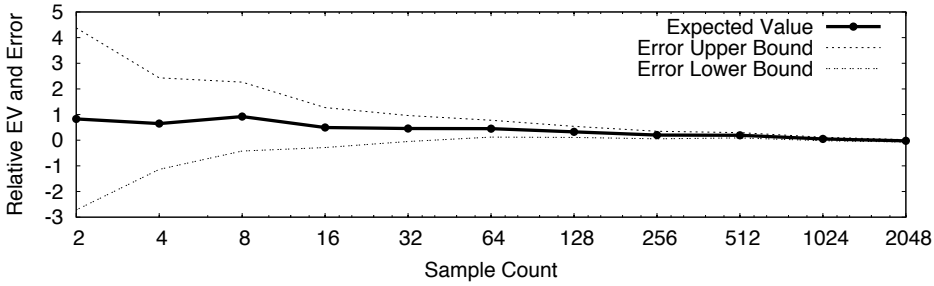


Figure 3.9: The error of Carat's energy model quickly decreases as Carat gathers more measurements.

anomalies by 41%, as can be seen in Figure 3.8. The increase for users without any anomalies is only 10%, and may be attributed to users learning how to manage their battery better. This increase may or may not be a result of using Carat.

3.3.7 Validation

We validate the use of *rates* as a proxy for energy drain using the Monsoon Power Monitor[6] energy measurement device. PIII shows that the error of Carat's method when compared to hardware energy measurements is less than 0.0009%/s. This answers RQ3. — battery level change within a time period can be used as a reliable estimate for energy use.

When we apply our method to data with an increasing number of data points, we need to ensure that more data brings about better results. Having more data could increase the variance of the data, and therefore also the error bounds, until the results are no longer statistically significant. Figure 3.9 alleviates these concerns. The error in Carat's energy model for an application or device decreases to about 1 EV from the mean energy use of the model within 32 samples, and quickly reduces after that. The error is minimal at a few hundred samples. Since Carat collects a sample every time the battery level of the smartphone changes, it will have an accurate picture of the energy use of the device after 100%-300% of the battery has drained since the user started using Carat. This can mean up to a week in wall-clock time.

To validate that Carat can find energy anomalies, PIII explains how we created an artificial energy bug in the mobile Wikipedia application. The

[6] `https://www.msoon.com/LabEquipment/PowerMonitor/`, visited November 17, 2014

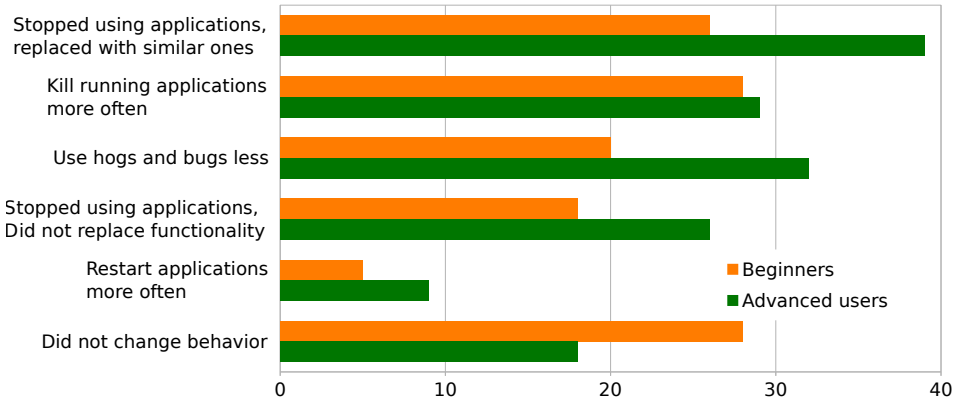


Figure 3.10: The questionnaire responses show some possible reasons why advanced users, having used Carat for three months or longer, gained increased battery life.

average battery consumption with the buggy application was significantly higher than that of other Wikipedia users, and Carat was able to discover the energy bug. This answers RQ4, validating that Carat can find injected energy bugs.

3.4 User Experience

The user interface of Carat is simple, with the main focus on a list of actions that the application recommends to the user. This list can be seen in Figure 3.4. Actions are of the form "Kill application X" or "Restart application Y" or "Upgrade the OS". Each action is associated with a battery life benefit, in hours and minutes, and its error, in ± minutes and seconds. Carat recommends killing an application if it has been found to use more energy than the average of all applications in the community. These Hogs are also shown in the Hogs tab. Restarting an application is recommended if the application uses more energy on a few users' devices, but not for the majority of Carat users. These are Bugs, also shown on a dedicated tab. The rationale for this is that the application might be caught in a bad state, and restarting it might help to reset it to normal. Because of Carat's statistical nature, we cannot rule out user behavior or environmental factors as reasons for applications using more energy on a device. Therefore we also do not recommend more drastic measures than restarting the application.

We conducted a survey together with Athukorala et al. with 1,140 respondents [106]. To track user behavior changes, we recorded which applications were Hogs and Bugs, and which users were running them. We then correlated the time that users had used our collaborative mobile energy-awareness application with the likelihood that users stop running Hogs and Bugs, and found a positive correlation. The results showed that Carat users can be categorized into two groups, beginners and advanced users. Advanced users are long-term Carat users, having used the application for three months or longer. They changed their behavior as a result of using Carat significantly more often than beginners.

The questionnaire responses shown in Figure 3.10 indicate that advanced users kill running applications more often, stop using energy-hungry applications, and switch energy-hungry applications to lighter ones. If there is no replacement application available, these users tend to use the energy-hungry application less.

This work answers RQ7 and RQ8. Specifically, users who participated in the survey that had been using Carat for three months experienced improved battery life. They gained a better understanding of the energy use of their device and applications. The results indicated that long-term users had significantly improved battery life, removed bugs and hogs more often, and switched from energy-hungry applications to lighter ones. Finally, this work contains guidelines for energy-awareness applications, including:

1. Show Your Work. Carat shows the user which applications are draining the battery faster than others. Users seem to follow Carat suggestions more often when they understand how it works. This understanding helps the user trust the recommendations. This is why we recommend applications to expose to the user not just recommendations, but also the reasoning or data behind them.

2. Retain Long-Term Users. Prolonged use of an energy-awareness application should result in increasingly better battery life. In Carat, the community significantly increases this effect, as recommendations get increasingly accurate as Carat is used for a longer period. It is therefore important for collaborative mobile energy-awareness applications to remain interesting for long-term users.

3. Give clear, action-oriented instructions for improving battery life. The most popular feature of Carat was the "actions" tab. The actions were more popular than the hogs or bugs, even though they simply tell users to "kill" or restart running applications on the hogs and bugs lists. Energy-awareness applications should therefore give action-oriented

instructions that clearly tell the users how to accomplish the goal of better battery life.

4. Distinguish System Components. Each new version of a mobile operating system includes new system processes that the user should not terminate. As Carat recommends killing processes that spend more energy than others, we need to maintain a list of these system processes. This problem can be addressed through crowdsourcing by allowing users to flag suspected system applications. Energy-awareness applications should therefore distinguish system components from third-party applications when making diagnoses and recommendations.

Chapter 4

Discussion

The results presented in Chapter 3 allow this thesis to answer the research questions shown in Section 1.3. The following section discusses these answers.

4.1 Research Questions Revisited

The first research question, RQ1, asked whether it is possible to gather rich context information on any smartphone platform. Section 3.1 answers this question in the positive. At the date of publication, there were few integrated sensors in smartphones, and external sensors were the only option. This work created a multi-platform data gathering framework, with a plugin-based architecture that was later used in many smartphone related systems.

RQ2 looked for the best smartphone platforms for energy and context data gathering. Section 3.2 answers RQ2 with Android and iOS.

The answer to RQ3 is presented in Section 3.3.7, where the author compares the power consumption of applications when measured by Carat, and when measured by a hardware power measurement device, and concludes that the difference is small enough that the battery level is a reliable indicator of power consumption.

RQ4 asks whether a collaborative mobile energy-awareness application is capable of detecting artificially created energy bugs, placed on a small number of devices in the form of a modified application. Section 3.3.7 answers this in the positive, validating the methodology of Carat, and allowing examination of RQ5 and RQ6.

RQ5 and RQ6 involve applications that have higher than normal energy use. RQ5 asks if a collaborative mobile energy-awareness application can detect Hogs, i.e. applications with higher energy use than other applications

on average. RQ6 asks if it can detect Bugs, i.e. applications that have normal energy use on average, but some users experience higher than normal energy use. The answer to both RQ5 and RQ6 is positive, described in Section 3.3.6.

The answers to RQ3–RQ6 make Carat useful for both smartphone users and researchers. Carat has also improved the battery life of its users, as described in Section 3.4. The Section answers RQ7, showing that Carat improves the battery life of its users by 10% to 41%, where larger gains are due to Carat helping pinpoint energy Hogs and Bugs on the device. Carat therefore has useful value to the user beyond collecting data and supporting research. Also, Long-term users of Carat gain significantly longer battery life and close running applications more often than new users. This indicates that Carat has helped them gain a better understanding of their device's energy behavior, answering RQ8.

4.2 Scientific Contribution

This thesis combines knowledge of rich context data gathering with smartphone systems, energy awareness, and collaborative data analysis. The author calls this work collaborative mobile energy awareness. Smartphone data gathering has become easier, and smartphone operating systems are improving their energy-awareness APIs at a quick pace. The results of applying the collaborative analysis method in the area of smartphone energy consumption are promising, and have made topics of research accessible.

This work has set the example for using the smartphone battery level instead of fixed or battery powered energy measurement hardware. This enables anonymous, large-scale power measurements of any features that researchers are interested in, on any smartphone. Large-scale data analysis is increasingly popular, often done by large companies and organizations. The author has shown that it can be also be used by a small research team. This should enable researchers to create projects much larger than before. The statistical approach is well suited to large-scale data analysis, since interesting thresholds can be determined by the data, instead of being set using expert knowledge or educated guessing. Being able to process the entire data instead of just a sample helps researchers get the complete picture and a more accurate model for the entire data set. In addition, planned future work is discussed in Section 4.4.

4.3 Practical Impact and Limitations

Our work is able to identify applications that use more energy than usual on Android and iOS smartphones. We can detect anomalies that use more energy on a small number of devices while behaving normally for most users, and we can identify applications that drain the battery quickly for all users. To do this, we only require the user to install a regular application from Google Play or the Apple App Store. The application consumes very little energy and informs the user about energy anomalies on their device within a week. This helps users gain insight into the energy behavior of their device and applications, and gives them the opportunity to control use of applications that drain the battery quickly.

Our work considers the same application running on many different types of devices identical. Therefore, batteries of different sizes and varying user behavior can introduce variance to the measured energy impact of an application. However, with enough data and a diverse enough community, these can be averaged out.

If an application is always running on a device, we cannot measure its energy impact, since there are no samples from that device without that application running. This includes system applications, such as the Android system process, the phone dialer, and email sync. Applications that always run together cannot be distinguished by Carat. For example, if Facebook and Twitter are always run together, but never one without the other, Carat cannot decouple their energy use from each other. However, if the application is run by many users, the difference in applications that the users run helps us single out the energy impact of these applications that are often run together.

4.4 Future Work

The Carat project and its dataset allow research into energy awareness, energy efficiency improvements, detecting and predicting infection by mobile malware, and investigation of trends in application use. We have barely scratched the surface of what can be done with the methodology and data gathered in this way. Deeper investigation in these areas is future work.

The impact of Carat on user behavior has been investigated at a coarse-grained level. At the time of writing, Carat has also gathered data on which buttons users click in the Carat application, and how much time they spend on each tab and screen of the application. The analysis of this data in conjunction with a user interface improvement study is future work.

Another item that demands further attention is the social aspect of Carat. While the application lets users see their applications and their impact on the battery, they have no quick way to share this information with the social groups they belong to. Being able to connect instances of Carat running on multiple devices would help, for example, family members see each others' battery life and decide whether it is a good time to call. The various energy bugs that your friends have could help you choose more energy-efficient applications for your daily life.

The current incarnation of the Carat project exposes to users recommendations to permanently close or restart applications, as well as upgrading the operating system. Energy bugs that are not caused by applications themselves, but application and device configurations, are not discovered with the current application. However, Carat has gathered the data required for such deeper analysis. We are investigating the impact of system settings on battery life on Android smartphones. This work will most likely be published in 2015.

We investigated the behavior of Carat users with a quantitative study with 1141 participants. M. von Kügelgen has conducted a qualitative study, with a group of interviewed participants who used Carat for a period of more than two weeks. The results of this study are to appear in her Master's thesis.

The application signatures gathered by Carat have been used to find mobile malware infection rates and develop a method to help find new mobile malware [117, 118]. In this work, we discovered that the definitions on what is malware vary between anti-malware vendors, and that the mobile malware infection rate for devices in the wild is higher than we expected, 0.22%–0.28%. We developed a method to estimate infection likelihood or vulnerability of devices to infection. This likelihood is visualized to the user as an intuitive pie chart of the device by a new application we have developed [119]. We may include this function into the Carat application in the future.

Chapter 5

Conclusions

This introduction has presented collaborative mobile energy awareness with the Carat project. The author has discussed the problem of gathering rich context and battery level data, and which smartphone platforms are best suited for the task. The Android and iOS platforms currently have the best support for collaborative mobile energy-awareness applications. We have discussed background information on energy awareness and statistical techniques. We have described related systems, and explained their relation to this work. In particular, previous work addresses the energy efficiency of individual sensors, communication media, and use cases of the smartphone. Some works target debugging of mobile applications, and a subset also energy debugging. Research has identified energy bugs as an important new phenomenon. Carat is the first collaborative approach to detect energy bugs.

The collaborative aspect of the methodology allows personalized analysis that goes deeper than data generated by any individual device. It can be used to determine whether energy use is normal for a particular application or device, by taking the community of users as the norm. In the Carat project, we collect battery level history data, and running applications, from hundreds of thousands of smartphones, and combine this data by application, by device, and their combination. We can then calculate the expected energy use of any given application across the user base of the project, or for a subset of devices running, or not running, an application. The accuracy of Carat depends on the error bounds of the data. From the statistics of these distributions we can detect applications, devices, and application-device combinations with anomalously high energy use. We call these anomalies Hogs and Bugs.

We have shown that collaborative mobile energy-awareness applications can be used to improve the battery life and user experience of smartphones.

Previous work has indicated that users are comfortable with following action-oriented recommendations, when given necessary background information, such as the reason why the action will improve battery life, and by how much. Users who had used Carat for three months saw a battery life increase of 41% on average, if they were running applications that Carat identified as anomalies. Users without any anomalies only gained a 10% battery life increase.

References

[1] C. H. (Kees) van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[2] Sasu Tarkoma, Matti Siekkinen, Eemil Lagerspetz, and Yu Xiao. *Smartphone Energy Consumption: Modeling and Optimization*. Cambridge University Press, August 2014.

[3] R. Trestian, A.-N. Moldovan, O. Ormond, and G. Muntean. Energy consumption analysis of video streaming to Android mobile devices. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 444–452, April 2012.

[4] Peter G. Bruce, Bruno Scrosati, and Jean-Marie Tarascon. Nanomaterials for rechargeable lithium batteries. *Angewandte Chemie International Edition*, 47(16):2930–2946, 2008.

[5] G. Girishkumar, B. McCloskey, A. C. Luntz, S. Swanson, and W. Wilcke. Lithium-air battery: Promise and challenges. *The Journal of Physical Chemistry Letters*, 1(14):2193–2203, 2010.

[6] Shuo Pang, J. Farrell, Jie Du, and M. Barth. Battery state-of-charge estimation. In *American Control Conference, 2001. Proceedings of the 2001*, volume 2, pages 1644–1649 vol.2, 2001.

[7] Seongjun Lee, Jonghoon Kim, Jaemoon Lee, and B.H. Cho. State-of-charge and capacity estimation of lithium-ion battery using a new open-circuit voltage versus state-of-charge. *Journal of Power Sources*, 185(2):1367 – 1373, 2008.

[8] O. Erdinc, B. Vural, and M. Uzunoglu. A dynamic lithium-ion battery model considering the effects of temperature and capacity fading.

In *Clean Electrical Power, 2009 International Conference on*, pages 383–386, 2009.

[9] M.V.J. Heikkinen and J.K. Nurminen. Measuring and modeling mobile phone charger energy consumption and environmental impact. In *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, pages 3194–3198, 2012.

[10] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 43–56, Berkeley, CA, USA, 2013. USENIX Association.

[11] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114, New York, NY, USA, 2010. ACM.

[12] Gerard Bosch Creus and Mika Kuulusa. Optimizing mobile software with built-in power profiling. In Frank H.P. Fitzek and Frank Reichert, editors, *Mobile Phone Programming*, pages 449–462. Springer Netherlands, 2007.

[13] A. Rice and S. Hay. Decomposing power measurements for mobile devices. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 70–78, 2010.

[14] P. Miranda, M. Siekkinen, and H. Waris. TLS and energy consumption on a mobile device: A measurement study. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 983–989, 2011.

[15] Zhenlin Guo, Wei Jiang, Nan Sang, and Yue Ma. Energy measurement and analysis of security algorithms for embedded systems. In *Proceedings of the 2011 IEEE/ACM International Conference on Green Computing and Communications*, pages 194–199, Washington, DC, USA, 2011. IEEE Computer Society.

[16] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones

with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42, New York, NY, USA, 2012. ACM.

[17] Andrius Aucinas, Narseo Vallina-Rodriguez, Yan Grunenberger, Vijay Erramilli, Konstantina Papagiannaki, Jon Crowcroft, and David Wetherall. Staying online while mobile: The hidden costs. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 315–320, New York, NY, USA, 2013. ACM.

[18] Aaron Carroll and Gernot Heiser. The systems hacker's guide to the galaxy energy usage in a modern smartphone. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pages 5:1–5:7, New York, NY, USA, 2013. ACM.

[19] Earl A. Oliver and Srinivasan Keshav. An empirical approach to smartphone energy level prediction. In *Proceedings of the 13th international conference on Ubiquitous computing*, pages 345–354, New York, NY, USA, 2011. ACM.

[20] Mohammad Ashraful Hoque, Matti Siekkinen, and Jukka K. Nurminen. Using crowd-sourced viewing statistics to save energy in wireless video streaming. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13, pages 377–388, New York, NY, USA, 2013. ACM.

[21] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer With Code Offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10)*, pages 49–62, New York, NY, USA, 2010. ACM.

[22] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 160–171, New York, NY, USA, 2002. ACM.

[23] G. Raffa, Jinwon Lee, L. Nachman, and Junehwa Song. Don't slow me down: Bringing energy efficiency to continuous gesture recognition. In *Wearable Computers (ISWC), 2010 International Symposium on*, pages 1–8, 2010.

[24] Nikodin Ristanovic, Jean-Yves Le Boudec, Augustin Chaintreau, and Vijay Erramilli. Energy Efficient Offloading of 3G Networks. In *Proceedings of the 2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, MASS '11, pages 202–211, Washington, DC, USA, 2011. IEEE Computer Society.

[25] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, 15(1):179–198, 2013.

[26] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring. In *USENIX Technical*, 2012.

[27] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 168–178, New York, NY, USA, 2009. ACM.

[28] Nilanjan Banerjee, Ahmad Rahmati, Mark D. Corner, Sami Rollins, and Lin Zhong. Users and batteries: Interactions and adaptive energy management in mobile systems. In John Krumm, Gregory D. Abowd, Aruna Seneviratne, and Thomas Strang, editors, *UbiComp 2007: Ubiquitous Computing*, volume 4717 of *Lecture Notes in Computer Science*, pages 217–234. Springer Berlin Heidelberg, 2007.

[29] Ahmad Rahmati and Lin Zhong. Human-battery interaction on mobile phones. *Pervasive and Mobile Computing*, 5(5):465 – 477, 2009.

[30] Mikko Heikkinen, Jukka Nurminen, Timo Smura, and Heikki Hämmäinen. Energy efficiency of mobile handsets: Measuring user attitudes and behavior. *Telemat. Inf.*, 29(4):387–399, November 2012.

[31] Davide Figo, Pedro C. Diniz, Diogo R. Ferreira, and João M. Cardoso. Preprocessing techniques for context recognition from accelerometer data. *Personal Ubiquitous Comput.*, 14(7):645–662, October 2010.

[32] Ahmad Rahmati and Lin Zhong. Context-for-wireless: Context-Sensitive Energy-Efficient wireless Data Transfer. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 165–178, New York, NY, USA, 2007. ACM.

[33] Nishkam Ravi, James Scott, Lu Han, and Liviu Iftode. Context-aware battery management for mobile phones. In *Proceedings of the 2008 6th Annual IEEE International Conference on Pervasive Computing and Communications*, pages 224–233, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Yu Xiao, Pan Hui, Petri Savolainen, and Antti Ylä-Jääski. CasCap: Cloud-Assisted Context-Aware Power Management for Mobile Devices. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 13–18, New York, NY, USA, 2011. ACM.

[35] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI*, 2013.

[36] Denzil Ferreira, Eija Ferreira, Jorge Goncalves, Vassilis Kostakos, and Anind K. Dey. Revisiting human-battery interaction with an interactive battery interface. In *Proceedings of Ubicomp 2013*. ACM, 2013.

[37] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, pages 10:1–10:14, New York, NY, USA, 2013. ACM.

[38] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 57–70, Berkeley, CA, USA, 2013. USENIX Association.

[39] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. There's an app for that, but it doesn't work. Diagnosing Mobile Applications in the Wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 22:1–22:6, New York, NY, USA, 2010. ACM.

[40] Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen. Contextphone: A prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, 4(2):51–59, 2005.

[41] Jason Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[42] Matteo Ferroni, Andrea Cazzola, Domenico Matteo, Alessandro Antonio Nacci, Donatella Sciuto, and Marco Domenico Santambrogio. MPower: Gain Back Your Android Battery Life! In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 171–174, New York, NY, USA, 2013. ACM.

[43] Aaron Schulman, Thomas Schmid, Prabal Dutta, and Neil Spring. Demo: Phone Power Monitoring with BattOr, 2011. ACM Mobicom 2011. Available at `http://www.stanford.edu/~aschulm/battor.html`.

[44] Ahmad Rahmati, Angela Qian, and Lin Zhong. Understanding human-battery interaction on mobile phones. In *Proceedings of the 9th international conference on Human computer interaction with mobile devices and services*, pages 265–272, New York, NY, USA, 2007. ACM.

[45] Ahmad Rahmati and Lin Zhong. Human–battery interaction on mobile phones. *Pervasive and Mobile Computing*, 5(5):465 – 477, 2009.

[46] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 84–93, New York, NY, USA, 2007. ACM.

[47] Antti P. Miettinen and Jukka K. Nurminen. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 4–4, Berkeley, CA, USA, 2010. USENIX Association.

[48] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43:51–56, 2010.

[49] Mads Darø Kristensen and Niels Olof Bouvin. Scheduling and development support in the scavenger cyber foraging system. *Pervasive and Mobile Computing*, 6(6):677–692, December 2010.

[50] Rich Wolski, Selim Gurun, Ra Krintz, and Dan Nurmi. Using bandwidth data to make computation offloading decisions. In *in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2008), High-Performance Grid Computing Workshop*, 2008.

[51] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314, New York, NY, USA, 2011. ACM.

[52] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953, 2012.

[53] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K. Nurminen, Matti Kemppainen, and Pan Hui. Can offloading save energy for popular apps? In *Proceedings of the seventh ACM international workshop on Mobility in the evolving internet architecture*, pages 3–10, New York, NY, USA, 2012. ACM.

[54] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 37–42, New York, NY, USA, 2000. ACM.

[55] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348, New York, NY, USA, 2011. ACM.

[56] Mikkel Baun Kjærgaard and Henrik Blunck. Unsupervised power profiling for mobile devices. In Alessandro Puiatti and Tao Gu, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 104 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 138–149. Springer Berlin Heidelberg, 2012.

[57] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 353–362, New York, NY, USA, 2012. ACM.

[58] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 36–36, Berkeley, CA, USA, 2012. USENIX Association.

[59] Denzil Ferreira, Anind K. Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: A study of battery life. In Kent Lyons, Jeffrey Hightower, and ElaineM. Huang, editors, *Pervasive Computing*, volume 6696 of *Lecture Notes in Computer Science*, pages 19–33. Springer Berlin Heidelberg, 2011.

[60] Jason Flinn and M. Satyanarayanan. PowerScope: a tool for profiling the energy usage of mobile applications. In *WMCSA*, 1999.

[61] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Mobicom*, 2012.

[62] Earl Oliver. The challenges in large-scale smartphone user studies. In *HotPlanet*, 2010.

[63] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. In *HotMetrics*, 2008.

[64] Amit Sinha and Anantha P. Chandrakasan. JouleTrack: A web based tool for software energy profiling. In *DAC*, 2001.

[65] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *IMC*, 2009.

[66] Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulos, and Feng Zhao. Energy-accuracy trade-off for continuous mobile device location. In *MobiSys*, 2010.

[67] R. J. G. Simons and A. Pras. The hidden energy cost of web advertising. Technical Report TR-CTIT-10-24, Centre for Telematics and Information Technology University of Twente, Enschede, June 2010.

[68] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[69] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[70] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[71] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[72] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *DSN*, 2011.

[73] A. J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken. Using correlated surprise to infer shared influence. In *DSN*, 2010.

[74] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[75] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[76] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium on Security and Privacy*, 1992.

[77] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, December 1999.

[78] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *NIST/NCSC*, 1997.

[79] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[80] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[81] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2005.

[82] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, 2005.

[83] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[84] Daniel T Wagner, Andrew Rice, and Alastair R Beresford. Device Analyzer: Large-Scale Mobile Data Collection. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):53–56, 2014.

[85] Hannu Verkasalo and Heikki Hämmäinen. A handset-based platform for measuring mobile service usage. *info*, 9(1):80–96, 2007.

[86] Y. Xie, H. Kim, D. O'Hallaron, M. Reiter, and H. Zhang. Seurat: A Pointillist Approach to Anomaly Detection. In *RAID*, 2004.

[87] David J. Malan and Michael D. Smith. Host-based detection of worms through peer-to-peer cooperation. In *ACM Workshop on Rapid Malcode*, 2005.

[88] Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. Big Data Processing in Cloud Computing Environments. In *Proceedings of 12th International Symposium on Pervasive Systems, Algorithms and Networks*, pages 17–22. IEEE, 2012.

[89] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[90] Jimmy Lin and Alek Kolcz. Large-scale Machine Learning at Twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 793–804, New York, NY, USA, 2012. ACM.

[91] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big Data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, May 2011.

[92] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. In *Proceedings of the VLDB Endowment*, pages 1481–1492. VLDB Endowment, 2009.

[93] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 15–28. USENIX Association, 2012.

[94] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical report, UC Berkeley, Jul 2011.

[95] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 295–308. USENIX Association, 2011.

[96] Apache Mesos: Dynamic resource sharing for clusters. Website. Available at `http://incubator.apache.org/mesos`. Visited on 27 Jul 2014.

[97] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Proceedings of Grid Computing Environments Workshop*, GCE '08, pages 1–10, 2008.

[98] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[99] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[100] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, OSDI '04, 2004.

[101] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[102] Apache Hadoop. Website. Available at `http://hadoop.apache.org`. Visited on 27 Jul 2014.

[103] Jian Lin, Li Zha, and Zhiwei Xu. Consolidated cluster systems for data centers in the cloud age: A survey and analysis. *Frontiers of Computer Science*, 7(1):1–9, 2013.

[104] Spark: Lightning-fast cluster computing. Website. Available at `http://www.spark-project.org`. Visited on 27 Jul 2014.

[105] Scala language. Website. Available at `http://www.scala-lang.org`. Visited on 27 Jul 2014.

[106] Kumaripaba Athukorala, Eemil Lagerspetz, Maria von Kügelgen, Antti Jylhä, Adam J. Oliner, Sasu Tarkoma, and Giulio Jacucci. How Carat Affects User Behavior: Implications for Mobile Battery Awareness Applications. In Matt Jones, Philippe A. Palanque, Albrecht Schmidt, and Tovi Grossman, editors, *CHI '14: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1029–1038. ACM, 2014.

[107] Wokje Abrahamse, Linda Steg, Charles Vlek, and Talib Rothengatter. The effect of tailored information, goal setting, and tailored feedback on household energy use, energy-related behaviors, and behavioral antecedents. *Journal of Environmental Psychology*, 27(4):265 – 276, 2007.

[108] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In Hans-W. Gellersen, editor, *Handheld and Ubiquitous Computing*, volume 1707 of *Lecture Notes in Computer Science*, pages 304–307. Springer Berlin Heidelberg, 1999.

[109] Paul Dourish. What we talk about when we talk about context. *Personal Ubiquitous Comput.*, 8(1):19–30, February 2004.

[110] Joonas Kukkonen, Eemil Lagerspetz, Petteri Nurmi, and Mikael Andersson. BeTelGeuse: A Platform for Gathering and Processing Situational Data. *IEEE Pervasive Computing*, 8(2):49–56, 2009.

[111] C. Perera, P. Jayaraman, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Dynamic configuration of sensors using mobile sensor hub in Internet of Things paradigm. In *Intelligent Sensors, Sensor Networks and Information Processing, 2013 IEEE Eighth International Conference on*, pages 473–478, April 2013.

[112] Sasu Tarkoma and Eemil Lagerspetz. Arching over the mobile computing chasm: Platforms and runtimes. *Computer*, 44(4):22–28, April 2011.

[113] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *HotNets*, 2011.

[114] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Sam Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Mobisys*, 2012.

[115] Amazon elastic compute cloud (Amazon EC2). Website. Available at `http://aws.amazon.com/ec2`.

[116] OpenStack cloud software. Website. Available at `http://www.openstack.org`. Visited on 27 Jul 2014.

[117] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. *CoRR*, abs/1312.3245, 2013.

[118] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 39–50, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee.

[119] Eemil Lagerspetz, Hien Thi Thu Truong, Sasu Tarkoma, and N. Asokan. MDoctor: A Mobile Malware Prognosis Application. In *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on*, pages 201–206, June 2014.

Included Publications

I

Research Theme A: Context-Awareness with Data Gathering

Research Paper I

Joonas Kukkonen, Eemil Lagerspetz, Petteri Nurmi, and Mikael Andersson

BeTelGeuse: A Platform for Gathering and Processing Situational Data

In *IEEE Pervasive Computing*, IEEE, 2009, vol. 8, no. 2, pp. 49-56

Contribution: The BeTelGeuse project was started by Petteri Nurmi. The author co-developed the system with J. Kukkonen. The author did roughly half of the design, implementation, testing, and analysis of the work. The author was the lead author of the publication with P. Nurmi and J. Kukkonen.

BeTelGeuse:
A Platform for Gathering and Processing Situational Data

BeTelGeuse is an extensible data collection platform for mobile devices that automatically infers higher-level context from sensor data. The authors introduce the BeTelGeuse architecture and evaluate its impact on mobile phone performance.

Joonas Kukkonen, Eemil Lagerspetz, Petteri Nurmi, and Mikael Andersson
Helsinki Institute for Information Technology HIIT

The widespread availability and portability of mobile phones has led them to become the de facto platform for ubiquitous computing. As mobile phones' battery life and capabilities continue to grow, they're supporting increasingly complex applications that leverage information about a user's situation—their location, activity, and so on. Modern smart phones are especially well-suited to this task because they're often integrated with sensing devices that facilitate obtaining detailed and meaningful descriptions of a user's situation. For example, smart phones can use accelerometers and microphones to accurately determine user activity[1] and can use Global System for Mobile Communications (GSM), WiFi, and GPS capabilities to determine users' locations and provide meaningful descriptions of their situations.[2]

To facilitate the gathering and processing of sensor data, we've developed BeTelGeuse, an open source platform that supports collecting data from phone sensors, Bluetooth-enabled sensors, and Internet data sources. BeTelGeuse also infers higher-level context from sensor data, for example, by inferring user activity from accelerometer measurements using the activity plug-in. Contrary to existing tools, BeTelGeuse isn't limited to a specific runtime environment or to a specific set of sensors. We designed BeTelGeuse to be extensible, as well as easy to use and configure. It's freely available under the GNU Lesser General Public License from our Web site (http://betelgeuse.hiit.fi). In this article, we present BeTelGeuse's design goals and architecture and evaluate its performance.

Design Goals

Our design goals for BeTelGeuse are:

- *Multiplatform support.* As we discuss in the "Related Work" sidebar, existing platforms are typically limited to a specific runtime environment. This limits the studies that researchers can carry out because the study's participants will depend on the number of available devices. To enable large-scale studies, the data collection platform should run on different devices and runtime environments.
- *Extensibility.* New kinds of sensing devices and data sources are continuously becoming available so researchers must be able to easily extend the platform to support them. Moreover, the platform's sensor interface shouldn't be limited to a specific type of sensor connectivity, such as Bluetooth, 802.11, or integrated sensors.
- *Data accessibility.* A platform that collects context data can provide applications with context information so it should be easy to integrate it with applications and services. Moreover, if researchers use the platform to run user studies, they should be able to access data remotely without requiring access to the physical device.

Published by the IEEE CS ■ 1536-1268/09/$25.00 © 2009 IEEE

Related Work in Data Collection Platforms

We categorize existing data collection platforms based on the nature of data that they collect. Platforms that collect objective data are nonintrusive as they gather sensor data about users' actions and situations without user involvement. The advantages of objective data are that users don't have to be interrupted and data collection doesn't suffer from subjective interpretations or from recall failures. On the other hand, sensor data is often noisy and erroneous, and unable to convey meaningful information about the users' situational, motivational, or informational needs. To this end, many platforms increasingly support subjective data collection. The most common way to collect subjective data is to use experience sampling, that is, explicitly ask for user feedback at regular intervals or in specific situations.[1] One alternative is to automatically infer meaningful descriptions from sensor data.

Several researchers have developed various platforms that support objective data collection. Most of these platforms are limited to a specific runtime environment or to a specific set of sensors. For example, ContextPhone logs various phone events (phone and application usage, for example), but can be used only on Nokia S60 devices.[2] Platforms that support multiple runtime environments are typically limited to a specific set of sensors or data type. For example, Intel PlaceLab[3] is limited to location data and BeTelGeuse's earlier version supports only Bluetooth-enabled sensors.[4]

Several platforms that support objective and subjective data collection have been proposed. Most of these platforms only run on devices from the Microsoft Windows CE operating system family. The first such tool was the Context-Aware Experience Sampling tool (CAES), which runs on PDAs using the Microsoft PocketPC operating system.[5] However, the CAES tool is no longer supported (the project was last updated in 2003). The most comprehensive platform so far is MyExperience,[6] which supports a wide range of sensors, contains a comprehensive event mechanism, supports a variety of experience sampling modalities, and has been extensively tested. Unfortunately, MyExperience is restricted to mobile devices running the Windows Mobile operating system.

Frameworks that automatically infer higher-level contexts from sensor data have been proposed. These systems typically focus on a specific contextual variable, and they don't have generic data collection capabilities. Examples include Opportunity Knocks,[7] which focuses on location information, and the Context Recognition Network, which focuses on activity information.[8] Because these frameworks focus only on the detection of activities, they serve the same purpose as the plug-ins in BeTelGeuse.

BeTelGeuse's main advantages are its support for multiple platforms and that its sensing capabilities scale according to the client device's capabilities. Thus, researchers can use BeTelGeuse on most platforms, but the amount and nature of data collections depends on the target device's available sensors and capabilities. Additionally, BeTelGeuse isn't limited to merely logging data; it can automatically and nonintrusively infer higher-level context from sensor data. Although earlier platforms support (subsets of) these features, BeTelGeuse is the first platform to support all of them. One of BeTelGeuse's limitations is that it currently doesn't contain built-in experience sampling functionality, but we're working on a plug-in for that.

REFERENCES

1. S. Consolvo and M. Walker, "Using the Experience Sampling Method to Evaluate Ubicomp Applications," *IEEE Pervasive Computing*, vol. 2, no. 2, 2003, pp. 24–31.
2. M. Raento et al., "ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications," *IEEE Pervasive Computing*, vol. 4, no. 2, 2005, pp. 51–59.
3. T. Sohn et al., "Experiences with Place Lab: An Open Source Toolkit for Location-Aware Computing," *Proc. 28th Int'l Conf. Software Engineering* (ICSE 06), ACM Press, 2006, pp. 462–471.
4. P. Nurmi et al., "BeTelGeuse—A Tool for Bluetooth Data Gathering," *Proc. 2nd Int'l Conf. Body Area Networks* (BodyNets), ACM Press, 2007; http://portal.acm.org/citation.cfm?id=1460232.1460253&coll=GUIDE&dl=GUIDE&CFID=25695462&CFTOKEN=85841578//.
5. S.S. Intille et al., "A Context-Aware Experience Sampling Tool," *CHI 03 Extended Abstracts on Human Factors in Computing Systems*, ACM Press, 2003, pp. 972–973.
6. J. Froehlich et al., "MyExperience: A system for in situ Tracing and Capturing of User Feedback on Mobile Phones," *Proc. 5th Int'l Conf. Mobile Systems, Applications and Services* (MobiSys), ACM Press, 2007, pp. 57–70.
7. D.J. Patterson et al., "Opportunity Knocks: A System to Provide Cognitive Assistance with Transportation Services," *Proc. 6th Int'l Conf. Ubiquitous Computing* (UbiComp 04), LNCS, Springer, vol. 3205, 2004, pp. 433–450.
8. D. Bannach et al., "Distributed Modular Toolbox for Multi-Modal Context Recognition," *Proc. 19th Int'l Conf. Architecture of Computing Systems* (ARCS 06), vol. 3894, LNCS, Springer, 2006, pp. 99–113.

- *High-level context*. Existing platforms are limited to gathering raw sensor measurements rather than inferring high-level abstractions of the user's location. High-level abstractions are often more meaningful and provide better clues about the user's actual situation, motivation, and information needs.

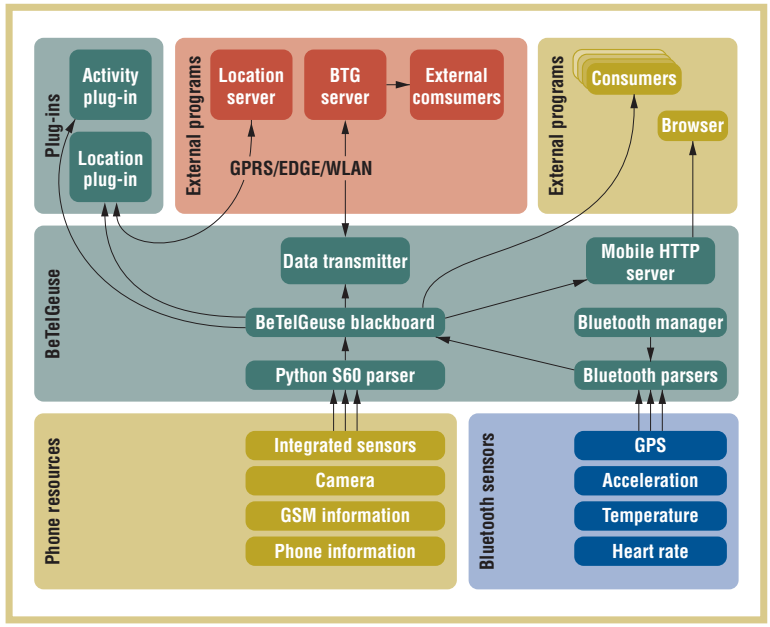


Figure 1. BeTelGeuse's high-level architecture. BeTelGeuse follows the microkernel architecture pattern.

- *User experience*. BeTelGeuse is aimed at two interconnected user groups: researchers who run user studies and extend BeTelGeuse by writing custom parsers or plug-ins, and users or study participants who run BeTelGeuse on their personal devices as part of a study or for their personal use (GPS traces or heart-rate data, for instance). From a researcher's perspective, user experience implies that BeTelGeuse should be easy to configure. For study participants, user experience refers to the platform's generic usability and that the tool doesn't have a noticeable impact on the client device's performance.

We describe BeTelGeuse's architecture in more detail.

BeTelGeuse Architecture

BeTelGeuse's high-level system structure is inspired by the microkernel architecture pattern. We have a separate core that offers the smallest set of functionality needed to run the tool. The core also defines interfaces for components that provide extended functionality. This allows a single implementation of the main functionality and custom extensions for different runtime environments. Our design has also been inspired by other context-aware frameworks. Similar to widgets in the Context Toolkit,[3] parser components act as abstractions of sensors, and we use a blackboard architecture inspired by the work of Terry Winograd.[4] Figure 1 shows BeTelGeuse's high-level architecture. The BeTelGeuse core contains data gathering, Bluetooth discovery, parser interfaces, the blackboard, and data transmission classes.

Implementation

To fulfill our goal of multiplatform support, we implemented BeTelGeuse's core using Java Micro Edition. We only use features that are commonly available in the Java implementations of most mobile and desktop devices. More specifically, BeTelGeuse's core is compatible with mobile systems that conform to the Mobile Information Device Profile (MIDP) 2.0 (http://jcp.org/aboutJava/communityprocess/final/jsr118) and Connected Limited Device Configuration (CLDC) 1.1 (http://jcp.org/aboutJava/communityprocess/final/jsr139) specifications, which are based on Java 1.3 and make BeTelGeuse compatible with desktop systems. Additionally, BeTelGeuse requires a JSR-82-compliant (http://jcp.org/aboutJava/communityprocess/final/jsr082) Java Bluetooth stack.

These devices satisfy BeTelGeuse's platform requirements:

- Mobile phones that support Java and Bluetooth, such as second and third editions of various Nokia S60 devices and Sony Ericsson W800i devices. (For a list of more than 100 compatible devices, please see http://developers.sun.com/mobility/device/pub/device/list.do).
- GNU/Linux PCs that run a standard Java installation with the freely available AvetanaBluetooth Java Bluetooth stack installed (http://sourceforge.net/projects/avetanabt).
- Windows PCs that run a standard Java installation and the freely available Bluesock Java Bluetooth stack (https://bluesock.dev.java.net).
- PDAs with Microsoft Windows Mobile running IBM J9 Java Virtual Maching with a commercial version of AvetanaBluetooth. We've tested BeTelGeuse on a Hewlett Packard hx4700 PDA running Microsoft Windows Mobile 2003 2nd Edition.

On smartphones, MIDP-specific socket connection classes are plugged into the core. For GNU/Linux and Windows, we used Java 1.5 socket classes. The Bluetooth parsers remain the same across platforms. We've included platform-specific parsers depending on the device that BeTelGeuse is deployed on—we enabled Python S60 extensions, for example, on Nokia S60 smartphones. BeTelGeuse's extensibil-

TABLE 1
Sensors currently supported by BeTelGeuse.

Sensor	Examples of measured data	Examples of events
Bluetooth GPS (NMEA 0183) sensors	latitude, longitude, altitude, time, number of satellites	LatitudeChange, LongitudeChange, timestamp, value equal/greater/smaller than a specified threshold
Alivetec Heart Monitor (www.alivetec.com)	ECG, 3-axis acceleration	ecgChange, accelerationChange, value equal/greater/smaller than a specified threshold
I-CubeX (http://infusionsystems.com)	distance (ultrasound), 3-axis acceleration, temperature, humidity, orientation, background light	value change events, value equal/greater/smaller than a specified threshold
Python S60 Parser (Nokia S60 3rd edition phones, requires signed Python)	GSM cell information: identifier, area, network and country codes, network name, signal strength. Call and SMS data: outbound and inbound calls and SMS, SMS access times. Phone status information: battery level, phone profile	value change events, value equal/greater/smaller than a specified threshold, callStart, callEnd, callAnswer, msReceive, smsOpen, smsSend, profileChanged, batteryLow
Local device	Bluetooth proximity information (In periodic scanning mode)	deviceAddressPresent, deviceNamePresent, deviceLost
Core	BeTelGeuse internal events source	parserCreated, parserDisconnected, parserDeviceLost, parserModeStreaming, parserModeRequest

ity also makes it possible to integrate platform-specific tools, such as MyExperience or ContextPhone.

Configuration

BeTelGeuse loads parameter values from a configuration file on startup. The configuration specifies Bluetooth mappings, frequency of data polling on each sensor, and whether to send data to a server or save it on the device. Users can modify the configuration through a MIDlet user interface. Alternatively, researchers or the study participants can specify a custom configuration file. We're currently implementing support for modifying the configuration remotely via command messages to the BeTelGeuse blackboard.

Blackboard

The BeTelGeuse blackboard acts as a hub for communications between different components and lets external components access the blackboard, such as when providing or receiving context data. BeTelGeuse Java plug-ins connect to the blackboard using direct method calls, whereas external components and plug-ins must use a local socket.

The blackboard uses a Simple Sensor Interface-like protocol (SSI; www.ssi-protocol.net). The messages begin with a command code, and most have component-type, user-id, and component-id following the code. The blackboard confirms command messages, but not data packets. The command code identifies the message. For example, "c" identifies a create message, which results in the blackboard creating a receiver and data container for the caller. The component-type, component-id, and user-id specify the message's target components (a subset or all of the components). This is useful in scenarios in which a number of components want to establish a dialogue. The current protocol version lets external components create, delete, and list components and components' owners, and download or upload data.

Data on the blackboard resides in memory. The blackboard is data-type agnostic and views the data as an opaque string of bytes. Components reading the data are responsible for interpreting it. By default, blackboard components interact in a publish–subscribe communication pattern. When a component receives new data, it notifies the blackboard, which, in turn, notifies consumers, that is, other components that have subscribed to the data. Each new data packet overwrites previous field values of the same component (GPS coordinates override old ones, but a longitude value only overwrites the old longitude value). Components might subscribe to receive data whenever a specific event occurs. For example, the GPS can be read at regular intervals or whenever the GSM cell changes. If a component subscribes to data changes, it's only notified when the data changes in the specified magnitude. Table 1 lists other supported event types. Components aren't required to subscribe to events; for example, parsers produce data but don't consume data produced by other blackboard components.

Data Transmitter

Although programs on the client device can access context data via the blackboard, remote researchers or external applications can't access data this way. To achieve data accessibility, BeTelGeuse contains a data transmitter, which synchronizes data with remote or local persistent storage and

makes it available to external components. Web applications can access data using the mobile HTTP server.

We implemented remote storage using a server-side component that stores the context data into a MySQL database. The data transmitter sends data using any Internet connectivity method that the client device supports—3G, GPRS, or wireless LAN, for example. The communications use a lightweight protocol, implemented on top of TCP. The protocol borrows ideas from existing sensor protocols, especially the SSI protocol, which is well-suited for communications between sensors and a controlling device. However, the SSI protocol is insufficient for our purposes because it doesn't contain messages for sending sensor names and identifiers, sending incremental sensor information, establishing a persistent session, or reconnecting to an existing session.

When local storage is used, data is stored on the device in a sequential file. The file resembles a data transmission log and the data transmitter can upload it to the BeTelGeuse server when Internet connectivity is available. Currently we don't support automatic replay of the transmission log, but the file can be uploaded manually. Internet connectivity rapidly drains the client device's battery life, but modern mobile devices support memory cards with a capacity of several gigabytes, so we can store several months of data locally.

Bluetooth Manager

The Bluetooth manager scans for Bluetooth devices and manages connections to Bluetooth sensors. The scanning can be performed periodically or initiated manually. Users can configure the scan interval using the MIDlet user interface or through remote access. Scanning in periodic mode is advantageous because it enables collecting (Bluetooth) proximity data, such as for social network analysis.[5] The periodic mode facilitates sensor management with stationary sensors scattered around the environment. In manual mode, Bluetooth scanning is performed at startup, after which users must manually trigger the scans using the MIDlet interface. This mode is useful when the sensor configuration doesn't change and proximity data isn't needed. Manual mode also helps avoid Bluetooth usage conflicts between scans and sensor data transmissions. On certain devices, such as older Nokia S60 second edition smartphones, Bluetooth scans require exclusive access to the Bluetooth stack, which can cause the receiving Bluetooth buffer to overflow with sensor data. Manual mode also slightly improves battery life.

The Bluetooth manager automatically connects to devices that users specify in the configuration and instantiates appropriate parsers. A device is specified by its (partial) friendly name (contains "GPS," for example) or Bluetooth address. Users can add new devices using the MIDlet interface or by editing the configuration file. When a Bluetooth scan finishes, the Bluetooth manager connects any matching discovered devices and creates appropriate parsers.

Connections to sensor devices might be lost for various reasons: wireless communication might be blocked, nearby devices can cause interference, sensor batteries might fail, or the sensors might become unreachable as users move. When a sensor connection is lost, the Bluetooth manager tries to restore the connection. The reconnection mechanism is based on an exponential back-off scheme. After a user-configurable maximum timeout is reached, the Bluetooth manager drops the sensor or continues the reconnection attempts with the maximum timeout, depending on the configuration.

Mobile HTTP Server

Because Web applications are increasingly based on locally executed JavaScript, we can easily enable Web applications to access context data. Our solution integrates a lightweight component, which reads HTTP requests and returns context data, into the BeTelGeuse core. Web applications that run on the device's browser can access context data using Ajax. We support `HEAD`, `GET`, and `POST` requests that follow the HTTP 1.0 specification. To minimize overhead and delays, the server simply reads the URL and query string from the HTTP request (/index.html?param=value) and returns sensor data. By default, the mobile HTTP server returns the context information in JavaScript Object Notation (JSON), which makes the information directly accessible as native objects and data structures for JavaScript code running on the device's Web browser. The mobile HTTP server also supports HTML and text formats. By default, all context data is returned, but the server can also be queried for a specific sensor's data. We use a query mechanism based on a Unix-style directory format. The URL, http://localhost/gps/latitude, for example, returns only GPS latitudes. Web applications can specify the data format separately as a query parameter: the call http://localhost/gps/latitude?format=html returns the latitude in HTML. Similar to the data transmitter and the BeTelGeuse server, the mobile HTTP server facilitates data access and contributes to our data accessibility goal.

Because Web applications are increasingly based on locally executed JavaScript, we can easily enable Web applications to access context data.

Context Parsers

Context parsers are abstractions of sensors that read and parse data, and write it to the blackboard. The parsers can operate in streaming or request mode. In streaming mode, data is continuously read from the sensor, whereas, in request mode, the sensor is polled for data when a certain event occurs or at user-configurable intervals. The request mode is useful for long-term data collection.

BeTelGeuse-compatible sensor types that can be used include external Bluetooth sensors, integrated phone sensors, software sensors, and Internet sensors. An Internet sensor could read Google Calendar entries, for example, and push the data to the blackboard. Developers can limit sensors to a specific platform. We use a platform-specific parser on Nokia S60 devices, for instance. Developers can also integrate BeTelGeuse with sensors using another communication technology. Table 1 lists the sensors BeTelGeuse currently supports.

To add support for new sensor types, developers must implement a new context parser or extend an existing one. Developers might also implement context parsers, written primarily in Java, as external plug-ins that push their data to the blackboard. When writing a parser for a new Bluetooth sensor, developers must assign the parser with an identifier and register it with the Bluetooth manager to ensure that the Bluetooth manager can automatically instantiate the parser. The BeTelGeuse Web site has details about implementing a parser and registering it with the Bluetooth manager.

Python S60 Parser

The Python S60 Parser provides information about phone status and usage on Nokia S60 devices and access to internal phone sensors. Currently, BeTelGeuse supports GSM information, call and SMS logging, and phone status information. The Python S60 Parser works similarly as other parsers. Thus, it registers to the blackboard, and other components can subscribe to data or events from the parser. In addition, it's possible to access phone calendar information or internal sensors, such as the microphone, camera, or integrated GPS.

Location Plug-in

The location plug-in consists of a server module and two client-side modules: a GSM positioning module and SerPens.[6] The GSM positioning module uses GSM fingerprinting to estimate the users' location whenever GPS is unavailable. SerPens is a collaborative tagging tool that lets users assign public and private labels to their locations. The labels are tied to a taxonomy that supports different granularities, such as country, region, and street. Users use private tags to indicate personally meaningful locations whereas public tags can be used to label landmarks. SerPens provides more meaningful data than mere coordinates, and can give clues about the context of the user. The location plug-in helps to achieve our high-level context information goal.

Researchers can develop new parsers, plug-ins, or Web-based extensions for BeTelGeuse.

Plug-ins and Extensions

Researchers can extend BeTelGeuse in various ways. They can develop new parsers or plug-ins. Additionally, they can develop Web-based extensions on top of the data transmitter.

Activity Plug-in

Whereas location typically provides clues about users' generic situation (home, work, school, and so on), activity information can provide clues about the environmental constraints that influence users' interaction. For example, while users are walking, they must pay attention to the environment, which often results in abrupt bursts and short-lived interaction patterns. Thus, mobile user studies could significantly benefit from detailed activity information. BeTelGeuse contains an activity plug-in that detects basic activities from accelerometer data. Our current implementation supports the Alivetec Heart Monitor (see Table 1) and detects the following activities: resting, walking, brisk walking, jogging, running, and commuting. We're currently modifying our plug-in to support the built-in accelerometers of recent Nokia smart phones and extending the number of identified activities.

BeTelGeuse Server

External applications or remote researchers running a multiperson study often need easy access to data from several devices. To facilitate data access, we've implemented a server component that maintains sessions with connected BeTelGeuse instances. It receives data sent by each instance and stores it in a MySQL database. The BeTelGeuse server uses a series of data filters for distributing data to external programs on reception. For example, the location server uses the BeTelGeuse server to obtain relevant location data (GSM + GPS).

Performance Evaluation

In analyzing BeTelGeuse's impact on client devices, we consider two aspects of performance: the memory requirements and the impact on battery life.

BeTelGeuse's memory footprint is between 5.6 and 7.3 Mbytes, depending on the configuration and the amount of sensors that are connected. The local version requires somewhat less memory than the online version. These figures include the memory required to run the Java virtual machine. In terms of instal-

TABLE 2
Battery lifetime in hours under different configurations.

#	Experiment setup	Mean	Standard deviation
1	Python	35.6	0.34
2	Python, Periodic GPS	34.1	0.44
3	Python, GPS Streaming	31.5	0.48
4	Python, GPS Streaming, BT Scan	25.0	0.24
5	Python, Server	6.0	0.15
6	Python, Server, Periodic GPS	5.8	0.28
7	Python, Server, BT Scan	6.2	0.05
8	Python, Server, Periodic GPS, BT Scan	5.8	0.54
9	Python, Server, GPS Streaming, BT Scan	5.0	1.04

lation size, BeTelGeuse requires only 179 Kbytes.

To measure the impact on the client device's battery life, we conducted a set of experiments in which we used BeTelGeuse under different configurations and measured the time it took to drain the battery of five fully-charged, brand-new Nokia E61i devices (with standard BP-4L 1500 mAh batteries). We considered nine different configurations and averaged the results over the devices. As we considered new devices, our results should be interpreted as upper bounds for performance. However, the homogeneity of the devices lets us draw better conclusions about the performance differences.

Table 2 shows our results. As our baseline, we used a version in which only the Python S60 parser collects data. This version lasted between 35 and 36 hours. Adding a GPS device that was read once per minute decreased the battery lifetime to 34 hours. Running Bluetooth scans on top of this had only a minor impact. These values span well over a day, which makes these setups well suited for long-term data collection. Changing the GPS from periodic reading mode to continuous streaming had a more significant effect on the battery lifetime, with the mean lifetime decreasing to 25.7 hours. Again, the Bluetooth scanning had only a minor impact on the performance (mean 25 hours). Thus, BeTelGeuse's battery usage is well-optimized with respect to Bluetooth.

The final experiments measured the effect of Internet connectivity on battery usage. In these experiments, we configured the transmitter to send data once every minute. As Table 2 indicates, the mean lifetime is roughly 5 to 6 hours, with the variation being caused by the amount of Bluetooth connectivity. For long-term data collection, the transmission rate should be decreased.

Case Studies

We used BeTelGeuse to collect large amounts of context data, and integrated it as a context source into a mobile application.

Gathering and Analyzing Location Data

We've used BeTelGeuse to collect GSM and GPS data from seven users for more than one month. The participants used a Nokia E61i mobile phone and an external Bluetooth GPS receiver. The GPS was polled every 60 seconds. We also scanned for nearby Bluetooth devices and gathered internal phone information with the Python S60 parser. An informal user study indicated that BeTelGeuse was easy to use, but participants considered the GPS receiver's short battery lifetime inconvenient. We've also used BeTelGeuse to collect GPS traces with different spatial and temporal characteristics from 10 to 15 different countries.[2]

Context-Adaptive Widgets

Capricorn is an adaptive, Web-based widget engine for mobile devices,[7] which uses BeTelGeuse as a context source and enables widgets to adapt their information based on the user context. For example, a context-aware travel planner can automatically fill in the origin of travel using location information provided by SerPens, and a news or weather service can provide localized information using location information provided by BeTelGeuse. As Capricorn is a Web application, it accesses context information through the BeTelGeuse mobile HTTP server on the device.

We're currently improving the data transmitter by adding support for data encryption and event-based connectivity. We're also extending BeTelGeuse to use new context sensors and continuing to develop additional plug-ins.

the AUTHORS

Joonas Kukkonen is a research assistant at the Helsinki Institute for Information Technology HIIT. His research interests include recommender systems and personalization of mobile applications. He is an MSc student in the Department of Computer Science at the University of Helsinki. Contact him at joonas.kukkonen@cs.helsinki.fi.

Eemil Lagerspetz is a research assistant at the Helsinki Institute for Information Technology HIIT and an MSc student at the University of Helsinki. His research interests include mobile data management and data communications. He has a BSc in computer science from the University of Helsinki. Contact him at eemil.lagerspetz@cs.helsinki.fi.

Petteri Nurmi is a researcher at the Helsinki Institute for Information Technology HIIT. His research interests include personalization of mobile applications and services, and mobile human computer interaction. He has an MSc in computer science from the University of Helsinki. Contact him at petteri.nurmi@cs.helsinki.fi.

Mikael Andersson is a research assistant at the Helsinki Institute for Information Technology HIIT and an MSc student in communications engineering at the Helsinki University of Technology. His research interests include computer networking with a focus on application layer services, and locationing and data-gathering on mobile devices. Contact him at mikael.andersson@tkk.fi.

Specifically, we're extending the activity recognition plug-in and developing an experience sampling plug-in that supports context-triggered questionnaires. Furthermore, we're planning to use BeTelGeuse in various context-aware applications as a context source.

REFERENCES

1. J. Lester, T. Choudhury, and G. Borriello, "A Practical Approach to Recognizing Physical Activities," *Proc. 4th Int'l Conf. Pervasive Computing* (Pervasive 06), vol. 3968, LNCS, Springer, 2006, pp. 1–16.
2. P. Nurmi and S. Bhattacharya, "Identifying Meaningful Places: The Nonparametric Way," *Proc. 6th Int'l Conf. Pervasive Computing* (Pervasive 08), vol. 5013, LNCS, Springer, 2008, pp. 111–127.
3. A.K. Dey, G.D. Abowd, and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human-Computer Interaction*, vol. 16, nos. 2–4, 2001, pp. 97–166.
4. T. Winograd, "Architectures for Context," *Human–Computer Interaction*, vol. 16, no. 2, 2001, pp. 401–419.
5. N. Eagle and A.S. Pentland, "Reality Mining: Sensing Complex Social Systems," *Personal and Ubiquitous Computing*, vol. 10, no. 4, 2006, pp. 255–268.
6. S. Bhattacharya et al., "SerPens—A Tool for Semantically Enriched Location Information on Personal Devices," *Proc. 3rd Int'l Conf. Body Area Networks*, ICST, 2008; http://portal.acm.org/citation.cfm?id=1460257.1460297&coll=GUIDE&dl=GUIDE&CFID=25695462&CFTOKEN=85841578//.
7. F. Boström et al., "Capricorn—An Intelligent User Interface for Mobile Widgets," *Proc. 10th Int'l Conf. Human-Computer Interaction* (MobileHCI 08), ACM Press, 2008, pp. 328–330.

Research Theme B: Mobile Operating System Energy Efficiency APIs

Research Paper II

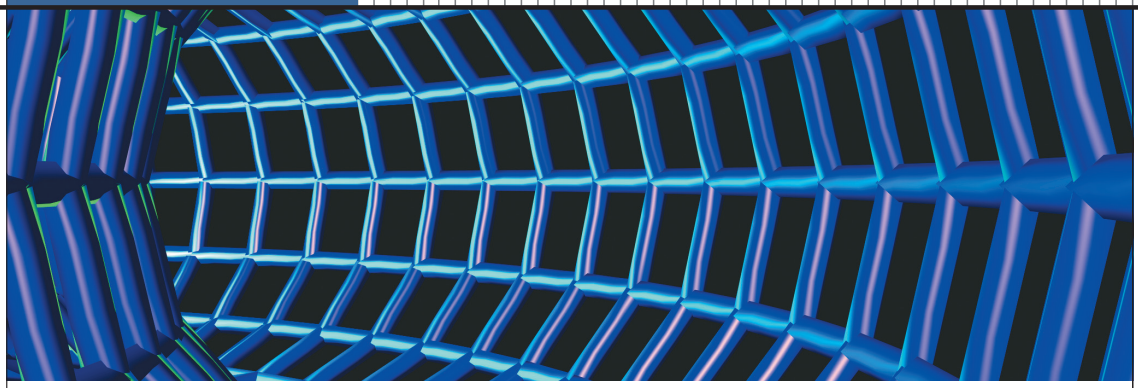
II

Sasu Tarkoma and Eemil Lagerspetz

Arching over the Mobile Computing Chasm: Platforms and Runtimes

In *Computer*, IEEE, 2011, vol.44, no.4, pp.22-28

Contribution: This publication is a survey of mobile platforms. The author was the lead author with S. Tarkoma. Particularly, the author gathered and wrote information on Java ME, Android, and Windows Phone 7, with special interest on network and energy monitoring.



Arching over the Mobile Computing Chasm: Platforms and Runtimes

Sasu Tarkoma and Eemil Lagerspetz, *University of Helsinki*

Platforms, runtimes, and middleware play a vital role in an evolving mobile computing environment in which the trend is toward converged communication, where Web resources integrate seamlessly with mobile systems.

Mobile devices increasingly depend on reliable software to offer a good user experience. Developing software in this operating environment requires many support services,[1-3] which are mainly provided in the middleware layer. Middleware offers a level of indirection and transparency for application developers, who save development cost and time using standardized or well-known interfaces when designing their products.

Development time and cost have traditionally been high for mobile applications and services, which operate in a more challenging environment than a typical fixed network. The wireless and mobile environment is less stable, has high latency, limited bandwidth, and many terminal types. Therefore, a specific implementation is not necessarily usable by all mobile equipment on the market.

Managing high development costs, meeting the challenges of the wireless network, and supporting device mobility motivate mobile handset manufacturers and vendors to provide middleware solutions for easier development and a unified user experience in the fragmented mobile marketplace.[4,5]

MOBILE MARKETPLACE EVOLUTION

In the 1980s, mobile phones provided only basic voice services.[6] The first generation of mobile applications and services, introduced around 1991, were restricted by technology. The two key enablers for application development were the mobile data connection and the Short Message Service.

The second generation of mobile applications was supported by built-in browsers, such as the Wireless Application Protocol (WAP) and, more recently, lightweight Web browsers. This generation also introduced the Multimedia Messaging Service (MMS) for images, audio, and video.

A more sophisticated environment supports third-generation applications and services that are built atop a platform offering services such as location support, content adaptation, storage, and caching. Platforms supporting the emergence of third-generation applications include Symbian Series 60, Java ME, Android, and the iPhone iOS.

Although the fourth generation of applications is still emerging, we can briefly sketch their anticipated properties in light of recent proposals in the research and standardization communities. The fourth generation is expected to

 Published by the IEEE Computer Society 0018-9162/11/$26.00 © 2011 IEEE

Table 1. Overview of popular smartphone systems.

Property	Android Linux	iPhone OS	Java ME MIDP	MeeGo Linux	Symbian Series 60	Windows Mobile .NET and Windows Phone 7
Development	Java, native code with JNI and C/C++	Objective-C	Java ME	C/C++, Qt APIs, various	C++, Qt, Python, various	C# and .NET, Silverlight, various
Network and energy monitoring/ control	Several APIs	Limited API support, battery monitoring since 3.0	No	Several APIs, native calls	Yes	Yes (limited in WP7)
Background processing	Yes (services)	No (yes for 4.0)	Yes (multitasking support in MIDP 3.0)	Yes	Yes	Yes, not supported for third-party applications in WP7
HTML5	Yes, support depends on version	Yes	N/A	Yes, support depends on version	Yes, future versions	No, expected in future versions
SIP API support	Yes, support depends on version	Extension	Extension	Yes	Yes	No, possibly in future versions
Open source	Yes	No	No	Yes	Yes	No
Third-party application installation	Certificate, Android market	Certificate, Apple App Store	Certificate	Certificate	Certificate	Certificate, WP7 apps marketplace

be adaptive not only in terms of application behavior and content, but also in the networking stack and wireless interface. Always-on connectivity, multimode communications, mesh networking, adaptive network interfaces, and physical communication media will be important features of future mobile computing devices.

MOBILE PLATFORMS

Table 1 gives an overview of the different mobile platforms and their properties. C, C++, and Java are currently the dominant programming languages for mobile devices. Network scanning and interface control functions, which have varying levels of support in mobile platforms, are important when an application needs to monitor and control the wireless communications. Background processing, which denotes the platform's multitasking capabilities, and energy and power monitoring and control—two important aspects of mobile platforms—are fairly well supported across platforms. Both multitasking and energy-management features vary from system to system. Memory management and persistent storage are well supported across the platforms, as is location information.

HTML5, the next version of HTML, is in development. The first public working draft of the specification was made available in January 2008, and completion is expected around 2012. Browser vendors are already implementing HTML5 features as they are defined. Some HTML5 features, including the WebSocket API, advanced forms, offline application API, and client-side persistent storage (key/value and SQL), can significantly improve current mobile Web applications. The iPhone platform has good support for HTML5.

The Session Initiation Protocol (SIP) is a key signaling protocol in 3G and 4G wireless access networks for session management.[7] Some platforms expose a SIP API to developers.

Three platforms are fully open source: Android, Maemo/MeeGo, and Symbian OS. The platforms have varying systems for supporting third-party application installation and execution. Execution of privileged system functions requires certification or other means of obtaining permission. The newer platforms are less fragmented, whereas older systems are invariably fragmented.

Android

The Android operating system and software platform for mobile devices is based on the Linux operating system. Android was developed by Google and the Open Handset Alliance, which includes more than 30 companies. The platform allows development of managed code using a Java-like language that follows the Java syntax, but does not provide the standard class libraries and APIs. Instead, it uses libraries and APIs developed by Google.

Figure 1 shows the Android architecture. It is based on the Linux kernel and a set of drivers for the various hardware components, such as display, keypad, audio, and connectivity. Android includes a set of C/C++ libraries for use by its various components. The Android application

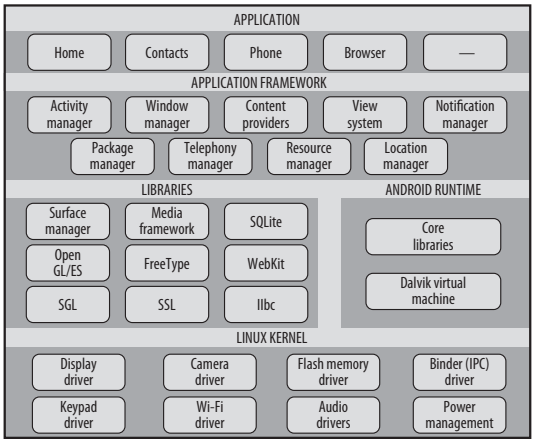


Figure 1. The Android architecture and its key components. Android is based on the Linux kernel and includes a set of C/C++ libraries, which are used by its various components.

framework APIs expose the capabilities of these libraries to developers.

The Android runtime executes the custom Java bytecode. The runtime includes the core libraries and the Dalvik virtual machine. Atop the libraries and runtime is the application framework, which consists of various managers. Several bundled applications reside atop the managers. These applications, which include an e-mail client, SMS program, calendar, maps, browser, and contacts, are all written in Java. Developers use the same API that the built-in core applications use. Android emphasizes component reuse, and any component can publish its capabilities, which other components can use if security constraints do not prevent this.

BlackBerry

RIM's BlackBerry devices are based on a proprietary operating system. Version 4 supports Java Mobile Information Device Profile (MIDP) 2.0 applications and synchronization with various productivity suites. The communications model is based on enterprise servers that act as e-mail relays. The servers use RIM's network operation center (NOC) to send and receive messages to and from the mobile devices. Because they use a proprietary NOC, the servers can implement mobile push efficiently.

iPhone

Apple developed the iPhone mobile operating system, or iOS, for its iPhone, iPod touch, and iPad products. The operating system is derived from Mac OS X and uses the Darwin foundation, built around XNU, a hybrid kernel combining the Mach 3 microkernel, elements of Berkeley Software Distribution (BSD) Unix, and an object-oriented device driver API (I/O kit).

Figure 2 gives an overview of the Mac OS X architecture, which was adapted for the iPhone architecture. The iPhone system is built on an ARM processor, and the core operating system (Darwin) includes the XNU kernel and system utilities. The XNU kernel includes Posix support, networking and file system support, and the device drivers. Above the operating system is the layered middleware—namely, core services, application services, the API layer, and finally the GUI (Aqua).

Apple provides the SDK as a free download, but requires approval and payment to release software for the iPhone platform in the App Store. There, users can browse and download applications directly to their iPhone, iPod touch, or iPad. Figure 2 shows the five available APIs: Carbon, QuickTime, BSD/Posix, Classic, and Cocoa.

Carbon is a procedural API consisting of a file manager, resource manager, font manager, and event manager. Each manager offers an API related to some functionality, defining the necessary data structures and functions. Managers are often interdependent or layered.

The Posix specifications define crucial operating system software interfaces and a standard threading library API.

The Classic environment, a backward-compatible hardware and software abstraction layer, is no longer supported in the current Mac OS version.

Cocoa Touch provides an abstraction layer based on Cocoa, the native Mac OS X object-oriented application program environment. Cocoa's design follows model-view-control (MVC) principles, and its frameworks are written in Objective-C. The Cocoa layer supports multitouch events and controls, and it has an interface for accelerometer input and support for localization (i18n) and a camera.

Announced in April 2010, version 4.0 of the iOS software supports multitasking for third-party applications. The key design principle is to offer APIs for specific background operations to optimize overall system performance. The new iPhone multitasking-specific APIs include support for background audio play, VoIP, location services, task completion, and fast application switching. For example, VoIP applications will be able to receive calls in the background. The APIs also support third-party push servers for sending notifications to applications.

The iPhone SDK supports the development of three types of applications—iPhone, iPad, and universal applications. A universal application determines the device type and then uses the available features based on conditional statements.

Java ME

Java Platform, Micro Edition (Java ME, previously J2ME), specifies a standardized collection of Java APIs for developing software for small and resource-constrained devices. Target applications include consumer devices, home appli-

ances, security, defense, automotive, industrial, industrial control, and multimedia. Since December 2006, the Java ME source code has been licensed under the GNU general public license.

A Java ME configuration specifies the virtual machine and the core libraries. There are two main configurations:

- the connected device configuration (CDC) for high-end PDAs, and
- the connected limited device configuration (CLDC) for mobile phones and other small devices.

Device manufacturers augment the configurations with profiles, which define additional APIs. The most common profile is the MIDP, aimed at mobile phones. The Personal Profile targets consumer products and embedded devices.

The Java ME platform's Mobile Service Architecture (MSA) specification (Java Specification Request [JSR] 248) defines a standard set of application functionalities for mobile devices, covering interactions between various technologies associated with the MIDP and CLDC specifications. An MSA version 2 device can use either CLDC 1.1 or CDC 1.1 as its configuration. The MIDlet execution environment is extended to CDC.

Java ME is evolving into a versatile platform for mobile application development. The introduction of MSA2 and various JSRs has gradually removed the early restrictions with MIDP applications and won growing vendor support. Moreover, MIDP version 3 addresses software portability challenges between CLDC and CDC.

Kindle SDK

Amazon offers a Kindle SDK for developing Java-based active applications for Kindle e-book readers. The Kindle SDK is based on the Java ME Personal Basis Profile and Kindle-specific extensions. The APIs support a basic user interface, networking, and limited secure storage on the device.

Maemo and MeeGo

Nokia's Maemo platform includes the Internet Tablet OS, which is based on Debian GNU/Linux and draws much of its GUI, frameworks, and libraries from the GNU Object Model Environment (Gnome) project. It uses the Matchbox window manager and, like Ubuntu Mobile, uses the GTK-based Hildon as its GUI and application framework. The Maemo platform is intended for Internet tablets, which are smaller than laptops but larger and more versatile than PDAs. A tablet might have a small keyboard, and its central characteristics include a stylus and a touch-sensitive screen. The touch screen is an important consideration for developers when designing graphical interfaces.

The latest development combines Nokia's Maemo platform with Intel's Moblin to form the MeeGo system. Both

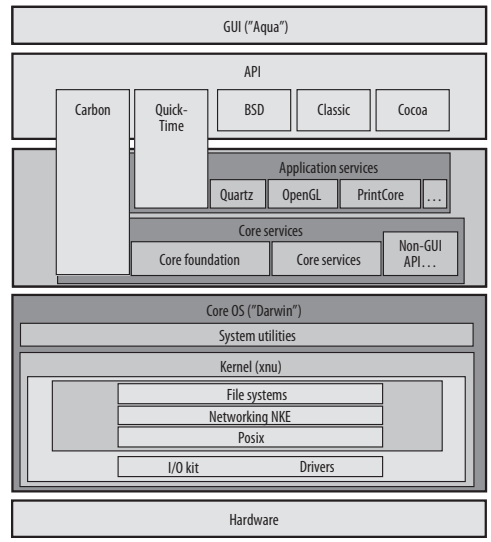


Figure 2. iPhone architecture and its main components. The architecture uses the Darwin operating system, which includes the XNU kernel and system utilities.

the Maemo and Moblin applications were developed with the GTK framework. However, Nokia's Qt framework has replaced GTK. MeeGo is expected to run on both Atom and ARM processors and to support both netbooks and mobile phones. MeeGo applications are written in C++ using the MeeGo SDK, which includes Qt. In February 2011, Nokia announced that its future smartphones will be based on the Windows Phone 7 platform.

The MeeGo architecture includes a hardware abstraction layer (HAL), an operating system base (Linux kernel, X), middleware, and user-experience-related functions. The lowest HAL layer, provided by the device vendor, includes kernel drivers and patches, kernel configuration, modem support, and other software related to the underlying hardware. MeeGo includes a set of components called the *content framework* to gather and offer user metadata to application developers.

Qt is a cross-platform application framework designed for building GUI applications. It provides the basic APIs for GUIs, databases, XML, and networking, and includes a WebKit-based Web runtime. The Qt platform is available for several systems, including Windows, Mac OS, Linux, Symbian, and Windows CE. The Qt API is implemented in C++, which most developers use. Currently, developers can only use C++ to create Symbian applications, although other language bindings are available for other platforms. The Qt platform is currently being extended to support device-specific APIs pertaining to location, calendars, alarms, sensors, and so on.

HP WebOS

HP's WebOS, originally developed by Palm, runs on the Linux kernel. The mobile runtime system includes a WebKit-based browser, and applications are written in JavaScript. The WebOS follows the cloud-based service model. The JavaScript-based application framework, called Mojo, provides common functions pertaining to user interfaces, widgets, and data access. A typical WebOS application uses HTML5 for presentation and audio/video. Developers create the applications using the MVC architectural pattern to separate user experience concerns from the application's data model and storage.

The WebOS is based on a scene metaphor in which an application consists of a set of scenes that facilitate presentation and user interaction. Scenes are pushed into and popped out of a scene stack. The top scene in the stack is visible to the user. The execution framework activates and deactivates the scenes. The scenes and applications use asynchronous notifications (W3C document object model events) to signal changes.

A mobile platform must be flexible and extensible not only in the distributed environment but also in the local environment.

Symbian and Series 60

Symbian's open mobile operating system is designed for ARM processors. The system includes a microkernel operating system, associated libraries, a user interface, and a reference implementation of common tools. Like many desktop operating systems, Symbian is structured with preemptive multitasking and memory protection. The multitasking model features server-based asynchronous access based on event passing. Three design goals motivated the choice of servers, microkernel design, and event passing:

- minimizing response times to users,
- maximizing integrity and security, and
- utilizing scarce resources efficiently.

Nokia acquired ownership of Symbian in 2008 and established the Symbian Foundation to provide royalty-free software for the mobile environment. The Symbian operating system was open sourced in 2010.

The base services layer is the lowest level reachable by user-side operations. It includes the file server and user library; the plug-in framework, which manages all plug-ins; a store; a central repository; a DBMS; and cryptographic services. The base services layer provides basic connectivity and serial communications as well as telephony. The communications infrastructure was developed on this layer, with two prominent networking stacks—TCP/IP and WAP. The Web and WAP browsers are available for the respective protocol stacks. The Symbian Web runtime is based on the WebKit system. The Java runtime and JavaPhone are available for applications.

The Symbian operating system's native language is C++, but the language is not compatible with ANSI C++. The operating system and applications are based on the MVC design pattern, which supports the separation of functions. The Symbian operating system emphasizes resource recovery using several programming features, such as a cleanup stack and descriptors. The operating system's event-based nature allows the minimization of thread switching using *active objects* that support asynchronous processing by encapsulating service request and request completion processing. In addition to C++ native applications, widgets are supported through the Nokia Web Runtime (WRT) widgets. The WRT environment follows the W3C widgets specification and allows widget installation and execution. The widgets can access device-specific features using the JavaScript Platform Services 2.0 API.

Windows Mobile and .NET Compact Framework

Microsoft released Windows Mobile 6 at the 3GSM World Congress in 2007. It comes in a standard version for smartphones, a version for PDAs with phone functionality, and a classic version for PDAs without phone features. Windows Mobile 6 is based on the Windows CE 5.0 operating system and integrates with Windows Live and Exchange products. Software development for the platform typically uses Visual C++ or the .NET compact framework. When native client-side functionality is not needed, software developers can use server-side code that is deployed on a mobile browser, such as Internet Explorer Mobile bundled with Windows Mobile.

The next version is the Windows Phone 7 Series (WP7) announced at the 2010 Mobile World Congress. WP7 focuses on user experience and does not support third-party software multitasking.

CURRENT STATE

The current platform landscape is heterogeneous, with several operating systems, programming languages, and interfaces in use, resulting in complex mobile software development and testing processes. A mobile platform must be flexible and extensible not only in the distributed environment but also in the local environment. The current and emerging platforms are still limited in this respect. For example, because third-party developers cannot easily extend Java ME MIDP, iPhone, or Android APIs, it is easier to extend and modify functionality at the server side than to modify the client.

The convergence of mobile and traditional IT fields has led to the increasing use of Web technologies in the development and deployment of mobile applications. The current Web technologies are suitable for mobile applications that conform to the Web's request/reply interaction style. However, in many cases Web protocols do not directly work well with mobile and wireless links. Indeed, asynchronous operation would be particularly useful in mobile applications that must react to changes in the environment.

Support for adaptive operation is an important trend in mobile applications and services. Adaptation can be realized in many ways—for example, on client devices or servers, using proxies and gateways, and through collaboration of the different entities, including services and software. Context awareness also introduces new challenges, such as context acquisition, privacy, and software testing and quality assurance. Testing adaptive and context-aware behavior requires new kinds of solutions and methods for ensuring that software is working properly and that it generates the desired user experience. Unfortunately, universal device and service discovery is still not available for developers. The current trend in developing adaptive applications is to use both Web technology and platform-specific APIs.

Thus, the current state leaves much room for improvement. No common APIs for network scanning and selection or network interface control exist. Background processing is supported on most platforms, but not all. Application-level energy awareness is far from ubiquitous. However, memory management, persistent storage, and location information are widely supported.

The marketplace has a clear need for a common API that unifies network connectivity, energy awareness, and the user experience. This should take a form that is easy to deploy on existing devices and most software stacks.

TOWARD COMMON APIS

One solution to the current challenge of fragmented device base and development tools is to provide common APIs for service and application developers. Indeed, both device manufacturers and telecom operators are actively involved in various API development and standardization efforts.

One key aspect is the development language and environment. Although recent experimental results suggest that JavaScript-based application platforms can be executed on Web browsers, several practical challenges pertaining to performance and browser limitations remain.[8] One challenge is determining how to allow a Web-based application to access local system variables, such as context variables. Security and privacy are paramount.

Web runtimes therefore must provide access to client-side platform APIs, such as the file system, geolocation, or camera. Previously, these APIs were exclusive to native applications. The industry is focusing on JavaScript and URL-based APIs to solve the API fragmentation problem. At least in theory, JavaScript APIs should be accessible to any content rendered by the Web runtime.

The Open Mobile Alliance, a key mobile standardization organization, bases its browsing specifications on Internet technology, but limits profiles for constrained resources and user interfaces of mobile devices. The GSM Association's OneAPI initiative aims to define a commonly supported API for mobile operators exposing network information to Web application developers.

The Open Mobile Terminal Platform group is pursuing a standardization activity that defines requirements and specifications for simpler and more interoperable mobile APIs.

The marketplace has a clear need for a common API that unifies network connectivity, energy awareness, and the user experience.

CHALLENGES

Mobile computing and software development face several important challenges. A key problem is fragmentation, which can occur on multiple layers and dimensions (the operating system, platform and middleware, service API layers, and so on). Currently, the available operating systems and platforms have differing programming conventions, interfaces, and software distribution solutions. This increases software development costs and slows down the software ecosystem.

In addition to fragmentation, the nature of the APIs and the features of the underlying platform they expose differ widely. Most systems expose certain underlying system features, some requiring authorization to access. For example, access to context information and networking services varies from system to system.

An asynchronous system-wide event bus is a basic solution for interconnecting various on-device components; however, there is no single standard for this. For example, Android and Java ME use Java-specific events, MeeGo uses D-Bus, and HP's WebOS uses W3C events. One trend is to use URI-based conventions for naming system resources and services. This approach is used extensively in Nokia Platform Services, WebOS, and other runtimes. An alternative, albeit more radical, solution to fragmentation is to use virtualization to execute the entire mobile application software stack.[9]

Energy consumption is one of the greatest challenges for current mobile devices. Energy and power continue

to remain the most limiting factors for the performance of mobile computing systems. Battery capacity does not increase as fast as the requirements. Internet and Web 2.0 services, in particular, consume vast amounts of energy, resulting in short battery lifetimes and, ultimately, poor user experiences. Current research challenges include how to support energy accounting and execute applications across mobile devices and cloud-based systems.[10]

A considerable amount of R&D has gone into solutions for different kinds of mobile and pervasive environments that support a wide variety of applications. However, the solution landscape is still fragmented. The next step to realizing the visions of pervasive computing is to support access to context information and enable more intelligent information processing on client devices.

Given that there are more than 3 billion mobile devices on the market today, with projections indicating that the number will approach 5 billion in the near future, the prospects for mobile applications, services, and middleware appear promising. Handling such a large number of users with widely divergent device types and characteristics necessitates developing interoperable and high-performance platforms as well as a highly scalable and available fixed infrastructure.

One step toward extensibility and universality is to employ a common interoperable message bus that supports component discovery, capability negotiation, and communications. Researchers have proposed message passing, publish-subscribe,[11] and tuple spaces as key components for mobile and pervasive software, but these ideas have not yet found their way into products and standardization. HTTP and runtime-specific APIs or local sockets are still the common denominator for communications and for enabling intradevice communications.

Although it has not yet been adopted on a large scale in the mobile marketplace, the HTML5 specification offers one approach for providing persistent storage and a satisfactory user experience. The iPhone iOS is pioneering the use of HTML5. It remains to be seen how fast other mobile platforms adopt this new specification. HTML5 in combination with custom JavaScript APIs would open a world of possibilities for developing portable and cloud-assisted mobile software.

Another approach is to use virtualization techniques to support multiple operating systems and platforms on the same hardware, possibly at the same time. Organizations could also use virtualization to enhance system security. This is a future technology still maturing for mobile devices.[9] C

References

1. M. Weiser, "Ubiquitous Computing," *Computer*, Oct. 1993, pp. 71-72.
2. A.K. Dey, "Understanding and Using Context," *Personal Ubiquitous Computing*, vol. 5, no. 1, 2001, pp. 4-7.
3. K. Raatikainen, H.B. Christensen, and T. Nakajima, "Application Requirements for Middleware for Mobile and Pervasive Systems," *ACM SIGMobile Mobile Computing and Comm. Rev.*, vol. 6, no. 4, 2002, pp. 16-24.
4. S. Tarkoma, ed., *Mobile Middleware: Architectures, Patterns, and Practice*, John Wiley & Sons, 2009.
5. E. Oliver, "A Survey of Platforms for Mobile Networks Research," *ACM SIGMobile Mobile Computing and Comm. Rev.*, vol. 12, no. 4, 2008, pp. 56-63.
6. K.M. Dombroviak and R. Ramnath, "A Taxonomy of Mobile and Pervasive Applications," *Proc. ACM Symp. Applied Computing* (SAC 07), ACM Press, 2007, pp. 1609-1615.
7. H. Schulzrinne and E. Wedlund, "Application-Layer Mobility Using SIP," *ACM SIGMobile Mobile Computing Comm. Rev.*, vol. 4, no. 3, 2000, pp. 47-57.
8. T. Mikkonen and A. Taivalsaari, "Creating a Mobile Web Application Platform: The Lively Kernel Experiences," *Proc. ACM Symp. Applied Computing* (SAC 09), ACM Press, 2009, pp. 177-184.
9. L. Rudolph, "A Virtualization Infrastructure that Supports Pervasive Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009, pp. 8-13.
10. E. Cuervo et al., "MAUI: Making Smartphones Last Longer with Code Offload," *Proc. Int'l Symp. Mobile Systems, Applications, and Services* (MobiSys 10), ACM Press, 2010, pp. 49-62.
11. P.T. Eugster et al., "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, 2003, pp. 114-131.

Sasu Tarkoma *is a full professor in the Department of Computer Science at the University of Helsinki. His research interests include mobile computing and Internet technology. He received a PhD in computer science from the University of Helsinki. Tarkoma is affiliated with the Nokia Research Center and Aalto University. Contact him at sasu.tarkoma@cs.helsinki.fi.*

Eemil Lagerspetz *is a researcher at the Helsinki Institute for Information Technology and a PhD student at the University of Helsinki. His research interests include mobile data management and data communications. Lagerspetz received an MSc in computer science from the University of Helsinki. Contact him at eemil.lagerspetz@cs.helsinki.fi.*

Research Theme C: Collaborative Energy Awareness

Research Paper III

Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma

Carat: collaborative energy diagnosis for mobile devices

In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys '13)*, ACM, 2013, Article 10, 14 pages

III

Contribution: A. J. Oliner started the Carat project with the idea to collect battery information from iPhone devices. The author co-authored Carat with Oliner, and came up with the a priori slice weighting for inaccurate iOS battery measurements, as well as the current version of the Carat algorithm, Algorithm 1. The author did roughly half of the design, implementation, testing, and analysis of the work.

Carat: Collaborative Energy Diagnosis for Mobile Devices

Adam J. Oliner, Anand P. Iyer, Ion Stoica
AMP Lab, UC Berkeley
{oliner, api, istoica}@eecs.berkeley.edu

Eemil Lagerspetz, Sasu Tarkoma
U of Helsinki
{eemil.lagerspetz, sasu.tarkoma}@cs.helsinki.fi

Abstract

We aim to detect and diagnose *energy anomalies*, abnormally heavy battery use. This paper describes a collaborative black-box method, and an implementation called Carat, for diagnosing anomalies on mobile devices. A client app sends intermittent, coarse-grained measurements to a server, which correlates higher expected energy use with client properties like the running apps, device model, and operating system. The analysis quantifies the error and confidence associated with a diagnosis, suggests actions the user could take to improve battery life, and projects the amount of improvement. During a deployment to a community of more than 500,000 devices, Carat diagnosed thousands of energy anomalies in the wild. Carat detected all synthetically injected anomalies, produced no known instances of false positives, projected the battery impact of anomalies with 95% accuracy, and, on average, increased a user's battery life by 11% after 10 days (compared with 1.9% for the control group).

1 Introduction

Mobile computing, especially smartphones and tablets, is becoming ubiquitous. Recent work [31] acknowledged the rise of a class of mobile software misbehavior: energy bugs. These bugs add to the list of causes of poor battery life that already includes system configurations, user behavior, and power-hungry apps. Significantly increased battery drain, called an *energy anomaly*, frustrates users, creates poor press for vendors, and can render devices unusable. For such a user, the goal is to understand what is using up the battery, whether or not that is normal, and what can be done.

For some devices, there are third-party apps and OS services for quantifying energy use and in some cases attributing it to specific processes [21]. Unfortunately, a single device has limited diagnostic power because there is no *a priori* specification of normal energy use (c.f. many correctness bugs; crashing is almost always bad). Local instrumentation alone is insufficient to determine whether observed energy use is normal or merely a consequence of local configuration parameters, system or device properties, or user behaviors. Without seeing the app running under different conditions, we cannot say whether changing some aspect of the system would improve battery life or by how much. No amount of local instrumentation can enable these capabilities; the information is simply not present on any single device.

We overcome this limitation by using a *community* of devices; ours is the first collaborative approach to energy diagnosis. Measurements aggregated from multiple *clients* allow us to collect more data more quickly, account (statistically) for individual variation in configurations and usage, say whether energy use is normal, and project the impact of certain actions. Each client occasionally records the battery level and other local data. We aggregate these measurements and compare average discharge rates under different conditions, such as which third-party apps (a common source of battery problems) are running.

If the average discharge rate while running some app A is higher than when A is not running (but any other apps may be), that app is an energy *hog*. A hog may be caused by a coding error (e.g., it prevents the screen from dimming) or because such energy use is intrinsic to the app's function (e.g., it frequently requires the GPS). If an app B is not a hog, it may be an energy *bug* on client X if the average rate on X is higher than the average on all the other clients running B. Energy bugs may be caused by a code error that only triggers under certain conditions (which our analysis tries to discover), configurations, or user behaviors. Distinguishing between hogs and bugs requires a collaborative method.

Our method for diagnosing energy anomalies uses the community to infer a specification (expected energy use), and we call deviation from that inferred specification an anomaly [9]. Unlike previous work, we are looking for regularity and deviation in the use of energy and leveraging this insight to characterize the abnormal use of that resource (the battery). Deviant energy use is an anomaly, regardless of the cause (e.g., coding error or user behavior). Our method further computes diagnosis trees called *MCADs*, which enable us to advise users what actions they can take to improve battery life and to estimate the amount of improvement (accompanied by error and confidence bounds).

SenSys'13, November 11–15, 2013, Rome, Italy.

Some prior work has aimed to understand energy use by employing a combination of hardware, OS, and app source code or binary instrumentation [11, 23, 32, 44]. In this paper, we present a non-invasive inference method for diagnosing energy anomalies that uses all the information available to a user app on both the Android and iOS platforms. In addition to being a pragmatic point in the design space, our solution naturally possesses several desirable qualities:

- Software-only. Hardware solutions are expensive, require technical skill, and void warranties.
- No kernel modifications. Hacking an OS requires skill; even "jailbreaking" may result in the user bricking their device or introducing bugs or security vulnerabilities.
- Black-box apps. The user does not have access to the source code for most of the apps they run or, usually, the ability to instrument binaries.

Extensions to our method could take advantage of platform-specific information (our implementation does so), but the aim of this paper is to evaluate how far we can take diagnosis without relying on such data. Distribution mechanisms like the app stores make it easy to get instrumentation onto off-the-shelf devices if that instrumentation is a standard app.

We take a black-box approach with process-level granularity; when we observe anomalously high energy use, we implicate one or more processes. Although this restriction may seem severe, for a method that can still be distributed via the App Store, our method is *maximally invasive*. Despite the limitations, these data are sufficient to diagnose anomalies with enough accuracy to provide actionable recommendations that improve battery life in practice.

In this paper, we do the following:

- Present a collaborative inference method for detecting and diagnosing energy anomalies by looking for deviation from typical battery use (see Section 2) and an implementation as an app called Carat for iOS and Android (see Section 3), and
- Evaluate our method with a 500,000-device deployment, showing a 100% detection rate of injected energy anomalies and partial corroboration for the thousands of anomalies we diagnosed in the wild (see Section 5).

The battery life of a device for which Carat generated action recommendations improves by an average of 41% during the first three months (compared with 7.9% for devices without Carat recommendations), 95.2% of the projected battery improvements (e.g., "Killing app A will increase battery life by 45m $\pm$ 5m") match the actual improvements within the 95% confidence bounds, and the battery overhead of running Carat is negligible (indistinguishable from running nothing, according to hardware power metering experiments). We conclude with a discussion of the limitations of our approach (see Section 6), an explanation of our place among the related work and how we distinguish ourselves (see Section 7), and a summary of the conclusions (see Section 8).

2 Method

Our method builds and compares conditional probability distributions of rates of energy use to look for *energy anomalies*; e.g., the rates when an app is running on a client with one OS version (the *subject distribution*) may be significantly higher than when running on clients with another OS version (the *reference distribution*). We focus on two kinds of anomalies: hogs and bugs (see Section 2.1). In Sections 2.2–2.4, we compute the magnitude of an anomaly, corresponding to the expected improvement in battery life that an average user experiencing the anomaly would see if they became like the average user not experiencing it. We quantify the error and uncertainty of these projected improvements and decrease that uncertainty by classifying measurements according to various conditions (e.g., rates taken when WiFi was, or was not, available). We generate the classifiers for an anomaly as a diagnosis tree (see Section 2.5–2.6), which we then reduce to a minimal, complete set of actionable recommendations (*MCAD*). An MCAD translates to anomaly diagnoses, such as "With C% confidence, killing app A would increase battery life by $d_1 \pm e_1$ minutes; upgrading to OS version V would increase battery life by $d_2 \pm e_2$ minutes; disabling WiFi…" and so on.

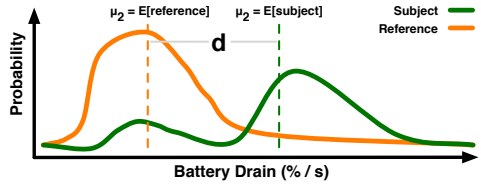


Figure 1. We compare the expected values of conditional distributions of energy drain rates to classify apps as hogs, bugs, or neither. The distance d shown is used to estimate the severity of the anomaly.

2.1 Hogs and Bugs

We define two categories of anomalies, hogs and bugs, by the types of subject and reference distributions we compare. Informally, an app is an energy hog when using that app drains the battery significantly faster, in a statistical sense defined in Section 2.4, than the average app. In contrast, an app has an energy bug when some running instances of the app (the ones in which the bug manifests) drain the battery significantly faster than other instances of the same app (the ones in which the bug does not manifest). Anomalies do not imply incorrect behavior; they may have innocuous causes. Hogs and bugs are computed as follows.

First, we build a (reference) distribution of battery discharge *rates* for devices used normally: playing games, browsing the web, making phone calls, leaving it idle, etc. Introduce an app A into the community, which some subset of clients will install and use, possibly in place of certain other apps. Build another (subject) distribution consisting only of rates observed while A is running. If the expected battery life while A is running is significantly lower than the expected lifetime without A, we call A an energy *hog*.

Intuitively, a hog lowers the community's average battery life. Note that an app may make use of energy-demanding device resources (e.g., WiFi or GPS) without being considered a hog; anomalous apps tend to overuse these resources.

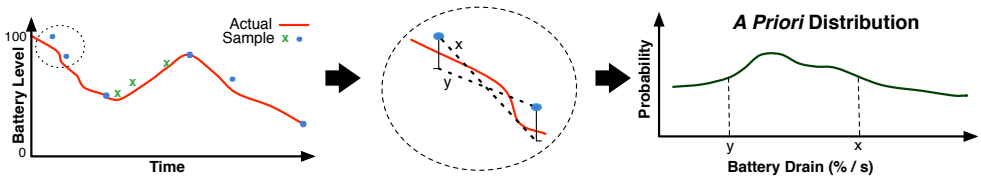


Figure 2. The process of converting battery level samples to rate distributions using the *a priori* distribution. Samples marked with green Xs are discarded because the device was charging. iOS may report a battery level up to 5% above the actual level. The slope bounds (x and y) determine the *a priori* slice.

An app could be a hog because of a coding error that affects many clients or because an app legitimately needs to use large amounts of energy to serve its function. Regardless, a user seeking to improve their battery life would do well to not have a hog running. Although per-device instrumentation, such as Android provides, can quantify energy use relative to other apps on one device, it cannot say whether that use is abnormal relative to other devices or to apps not running on the device, and so cannot detect or diagnose hogs.

An app B that is not a hog may still use much more energy on some client X. If the expected discharge rate of B running on client X (subject distribution) is significantly higher than that of B running on other clients (reference distribution), we call B an energy *bug* on client X. No amount of instrumentation on a single device can detect or diagnose bugs.

An energy bug is therefore a pair: an app and a client it afflicts. An energy bug may be caused by a coding error that affects a small group of clients, a rare configuration that uses more energy ("correct" or otherwise), or unusual user behavior (which requires a community to detect). If the buggy app is getting caught in a bad state, restarting the app may return the app to normal; otherwise, the remedy is the same as for a hog. Other actions may be suggested by our diagnosis trees (Section 2.6), but the current app UI does not reflect this.

We added a caveat that a hog cannot also be a bug to distinguish anomalies that affect all or most clients (hogs) from those that affect only a subset. Hogs are unlikely to be fixed by a restart, so we recommend killing them. This difference in appropriate response motivated the naming, and we found the distinction useful.

The subject and reference distributions are built using battery level samples from the community, as we explain in the following sections. The expected values of these distributions converge rapidly to the true expected value as the number of clients increases (see Section 5.7).

Note that even perfect knowledge of app behavior on a single client could not distinguish hogs from bugs; heavy energy use on one device could be a matter of configuration, user behavior, or some other bug trigger that stays static across runs. In order to say whether an app or app instance is anomalous, a community is required.

2.2 Conditional Distribution Model

As discussed in Section 2.1, to detect energy anomalies we compare two distributions of the battery drain (see Figure 1). This section explains how such a conditional distribution is modeled, and how we quantify the associated uncertainty. The input is a set of n *rates*, tuples consisting of a feature vector c and a rate probability *distribution* u, computed from some pair of samples (see Section 2.3). We model these as being randomly sampled from a true distribution U_c, with mean μ and variance σ^2, composed of measurements satisfying predicate c (e.g., iPhone 4 with WiFi access).

We first take the expected value of each u to yield a *rate* r. Consider the conditional distribution R_c of rates r satisfying c. To compute the error and confidence bounds on the expected value of R_c, we model it as n independent samples from U_c. These rates—means computed from a large number of random i.i.d. variables—are therefore approximately normally distributed as $\mathcal{N}(\mu, \frac{\sigma^2}{n})$, according to the Central Limit Theorem (CLT).

This result can also be obtained by starting with the assumption that R_c is distributed as $\mathcal{N}(\mu, \sigma^2)$. Although we do not know the parameters μ and σ^2, we can estimate them using the rates $(r_1, \ldots, r_n)$. The well-known maximum likelihood estimators for these parameters—obtained by maximizing the log-likelihood function—are as follows:

$$\hat{\mu} = \bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(r_i - \bar{r})^2.$$

By the Lehmann-Scheffé theorem, $\hat{\mu}$ is the uniformly minimum variance unbiased estimator for μ: $\hat{\mu} \sim \mathcal{N}(\mu, \frac{\sigma^2}{n})$.

This agrees with the CLT method. The estimator $\hat{\sigma}^2$, however, is biased, so we apply Bessel's correction to obtain the uniformly minimum variance unbiased estimator for the sample variance:

$$s^2 = \frac{n}{n-1}\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2.$$

By our normality assumption, we can construct the t-statistic $t = (\hat{\mu} - \mu)/(s/\sqrt{n})$, which has the Student's t-distribution with $n-1$ degrees of freedom. We can approximate the error bounds on this estimate of μ using a standard formula, where h is chosen according to the desired confidence level:

$$\mu \approx\in \left[\hat{\mu} - \frac{hs}{\sqrt{n}}, \hat{\mu} + \frac{hs}{\sqrt{n}}\right] = \hat{\mu} \pm \varepsilon$$

For 95% confidence error bounds, $h = 1.96$; we use this value for all experiments in this paper. Crucially, to esti-

mate the mean μ and to assign error and confidence bounds to that estimate, we require only the rates r, not the original distributions u.

As we gather more data, the uncertainty associated with these expected values decreases. We gauge empirically how convergence occurs in practice in Section 5.7.

2.3 Computing Rate Distributions

To compute rate distributions, our method must first convert a set of *samples* from a single client into a set of *rates*. A sample is a measurement taken at a particular point in time that consists of the battery level (%) and a list of features: device model, OS version, names of running processes, battery state (e.g., unplugged), etc. Let $s_t = (b, p, q, \hat{c})$ denote a sample taken at time t, triggered by reason q (e.g., the device was unplugged), where the battery level was observed to be at fraction $0 \leq b \leq 1$ and the battery state was p (e.g., unplugged). The remaining features are denoted collectively as a set $\hat{c}$ of key-value pairs (e.g., "OSVersion=5.0" or "AppXRunning=YES").

First, we sort the samples by t and filter them using the p values to retain only those adjacent samples that span a period during which the device was not plugged in, restarted, or otherwise increasing in battery level: that is, only periods when the battery was discharging. This reduces the initial set of all samples to a set of *consecutive pairs*. We compute discharge rates from these pairs.

Our method allows for imprecision in both the battery level and time measurements by converting a consecutive pair $s_{t_1} = (b_1, p_1, q_1, \hat{c}_1)$ and $s_{t_2} = (b_2, p_2, q_2, \hat{c}_2)$ not to a single rate number but to a rate distribution u. We associate this distribution with a set of features, yielding the pair $R = (u, c)$, computed from the features of the constituent pair of samples, as explained below.

If both endpoints, (b_1, t_1) and (b_2, t_2), are exact, then the rate distribution is $u = \frac{b_1 - b_2}{t_2 - t_1}$ with probability 1. Discharging yields a positive rate.

On iOS, we only get such exact measurements when the `UIDeviceBatteryLevelDidChangeNotification` is triggered. Otherwise, we estimate a probability distribution for the rate. There are a variety of techniques one might employ, depending on the nature of the uncertainty. In this paper, we address the case of iOS measurements, which present unique challenges. Specifically, the API provides battery level measurements at a granularity of 0.05. In other words, if we request the battery level at an arbitrary time during execution and get 0.95, the true level may be in the range $(0.90, 0.95]$.

The true rate, therefore, lies between $\frac{b'_1 - b_2}{t_2 - t_1}$ and $\frac{b_1 - b'_2}{t_2 - t_1}$, where $b'_1 = b_1 - 0.05$ and $b'_2 = b_2 - 0.05$, and subject to the constraint that the rate is nonnegative. Not all values in this range are equally likely, however, so we use this range to take a "slice" of an *a priori* rate probability distribution (see Figure 2), computed using the rates that clients were able to compute exactly, as described above. There was sufficient data in this distribution to bootstrap our method. We convert the slice to a probability distribution by dividing by the slice mass and use it as the rate distribution u.

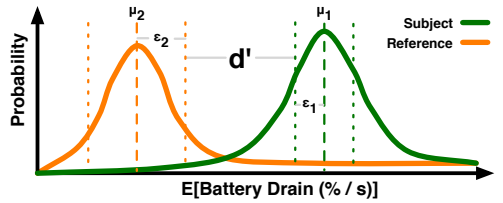


Figure 3. We compare distributions of the expected values of battery drain to identify anomalies ($d' > 0$) and quantify the error and confidence ranges for expected battery drain under different conditions.

We compute c from $\hat{c}_1$ and $\hat{c}_2$ by taking the union: $c = \hat{c}_1 \cup \hat{c}_2$. Features like device model do not change between consecutive samples. We conservatively say that an app was running during the period $[t_1, t_2]$ if it was seen in either sample. It would be straightforward to use a different function if the semantics of the features demanded it.

2.4 Comparing Rate Distributions

Let c_1 be the conditions of the subject distribution (e.g., app A is running) and c_2 be the conditions of the reference distribution (e.g., app A is not running). We aim to ascertain whether c_1 corresponds to *significantly greater* energy use than c_2. For this to be answered in the affirmative, we require the following:

$$\hat{\mu_1} - \frac{hs_1}{\sqrt{n_1}} - \hat{\mu_2} - \frac{hs_2}{\sqrt{n_2}} = \hat{\mu_1} - \hat{\mu_2} - (\varepsilon_1 + \varepsilon_2) \quad > \quad 0.$$

Otherwise, the data does not support the assertion with the desired confidence. Graphically, this corresponds to a positive value of d' in Figure 3.

Carat suggests actions that would improve battery life along with the expected value of that improvement for an average client (starting from full charge and fully draining the battery). The improvement if the client were to change from c_1 (experiencing the anomaly) to c_2 (not experiencing it) follows directly from the distance metric $d = \hat{\mu_1} - \hat{\mu_2}$. Within our confidence bounds, however, the value of d could be as much as

$$e = h\left(\frac{s_1}{\sqrt{n_1}} + \frac{s_2}{\sqrt{n_2}}\right).$$

This is symmetric about the expectation. The estimated improvement is therefore $d \pm e$.

2.5 Splitting Distributions

In order to more confidently diagnose anomalies, we build a tree that separates conditional distributions by features that significantly affect energy use. Let each conditional distribution be a node in this tree, uniquely identified by its condition c. Starting with some distribution c (e.g., app A is running), iterate through each feature $f \notin c$ and attempt a split by creating new child nodes $c \wedge f$ and $c \wedge \neg f$. For instance, if f is whether the client is running a Galaxy S II, then one child would get the rates from node c taken from Galaxy S IIs and the other would get all other rates satisfying c.

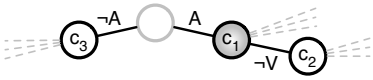


Figure 4. The minimal complete actionable diagnosis (MCAD) for the example anomaly c_1 described in Section 2.6, consisting of c_2 and c_3. The dashed lines indicate nodes and subtrees that, while produced via splits when the tree was constructed, did not meet the criteria for an MCAD.

Splitting has two competing effects on the error bounds. First, it reduces n, thereby increasing the error (increasing uncertainty). Second, if feature f divides rates from distributions having significantly different means, then it will likely reduce the sample variance of at least one child and thereby decrease the error (decreasing uncertainty).

A split is performed if the child nodes c_1 and c_2 yield a positive gap, $d' > 0$, as in Figure 3. Splitting generates two leaves, children of c, with edges f and $\neg f$. Otherwise, we make no changes to the tree and proceed to test the next feature. When no more features remain, we can recursively repeat the process on any new leaves.

2.6 Diagnosis

This section describes how to generate a diagnosis for an anomaly, which involves building a tree structure similar to a classification or decision tree [24, 39], and conclude with an example. Consider a node c_1 corresponding to a subject distribution for an anomaly (see Section 2.1). A *diagnosis* is a set of nodes with significantly lower energy use than c_1. Intuitively, a node in this diagnosis is some condition under which the anomaly does not occur. The diagnosis is *complete* if it includes all such nodes.

Let node c_2 be said to be *reachable* from node c_1 if, in the problem domain, it is possible to initially be in a state satisfying c_1 and, by performing some actions, then satisfy c_2. We define an *actionable* diagnosis to be one consisting only of reachable nodes.

A diagnosis is *minimal* if every subtree entirely contained in a complete diagnosis is replaced by its root. The minimal complete actionable diagnosis (MCAD) is unique, but note that it may include paths from c_1 to multiple different states.

For example, consider the node for running app A, $c_1 = A$, with significantly more energy use compared with $\neg A$; it is a hog. Say, for simplicity, that there are only two other features of the device—model M and OS version V—and only one other possible OS version. Every node in the subtree rooted at $\neg A$ has significantly lower energy use than c_1, as does every node with $\neg M$ or with $\neg V$. In our domain, a user cannot change their device model, so all nodes with $\neg M$ are excluded from the actionable diagnosis despite showing less energy use. To make the diagnosis minimal, replace with their respective roots the nodes in the subtrees rooted at $A \wedge \neg V$ and $\neg A$. Thus, the MCAD (illustrated in Figure 4) is exactly these two nodes (c_2 and c_3); the interpretation is that the client can improve their battery life either by changing OS versions or killing the hog.

These trees helped diagnose problems in the wild, such as the Kindle bug in Section 5.4.3 where WhisperSync was using far more energy when syncing over GSM. Our analysis discovered the bug was correlated with the iPhone 4 and only occurred on iPads when they did not have WiFi. There are dozens of such diagnoses that we have investigated, and in some cases reported to the developers, and thousands more produced by Carat.

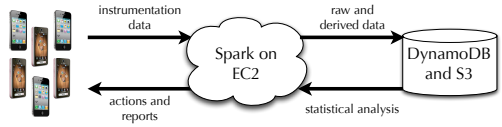


Figure 5. The Carat architecture, consisting of the crowd-based front end, the central server with the analysis running in the cloud, and the stored samples and results.

Although the client UI only displays recommendations to kill or restart an app or to upgrade the operating system, our analysis computes diagnoses—and can make recommendations based on—features like internet connectivity status (radio or WiFi), mobility, device model, app versions, GPS activity, the user ID (usually indicating a bad battery or strange user behavior), and so on. Thus, our MCADs can recommend actions like turning on/off the WiFi/GPS/radio, upgrading the app/OS to a newer version, or avoiding an app under certain conditions (e.g., while moving around or when not connected to the internet).

3 Implementation

The Carat architecture consists of a mobile app for device users (see Section 3.1), a central server that collects the data (see Section 3.2), and an analysis running in the cloud (see Section 3.3). Figure 5 shows an overview.

3.1 Carat App

We implemented Carat as an app on both the iOS and Android platforms. It is available as a free download on Apple's App Store, Google's Play Store, and as source code on GitHub, all of which are linked from the project homepage[1]. The clients are lightweight; e.g., the iOS app is ∼6000 lines of Objective-C, excluding third-party libraries like Flurry (for collecting usage statistics), ShareKit (for enabling sharing over social networks), Thrift (for handling messaging protocols), CorePlot (for plotting), and several others. This number also excludes auto-generated code related to the UI.

Carat runs as a user-level app on stock devices. This places platform-specific restrictions on what information is accessible and when our app is allowed CPU time to measure it. Our implementation records the following information using the public APIs:

- battery level fraction,
- battery state (e.g., plugged in or unplugged),
- names of running processes (each non-OS process roughly equates to a single user app),
- state of memory (e.g., number of active pages),
- OS and version,
- device model, and

- a unique, anonymous, Carat-specific client ID.

This information resides in persistent storage until the app is brought to the foreground, at which point it communicates with the Carat server over TCP. Our communication model is client-initiated (since they are situated behind NATs) and utilizes Apache Thrift to define the service interface.

The app intermittently transfers stored samples to the server over 3G or WiFi. Since we optimized Carat with respect to energy use, the client invokes a data transmission to the server only when it is running in the foreground and when the user is interacting with the UI. At this time, the app also requests results from the server to update the UI.

To comply with legal restrictions and to alleviate user concerns, our implementation neither records nor transmits personally-identifying information. What it does record is visible within the app (see Section 3.1.1), so the user knows exactly what Carat is measuring. Furthermore, our EULA (required by the App Store and also available on the project webpage[1]) includes an additional clause making it clear exactly what our app will do. Finally, the app is open source under a BSD license and is available on GitHub[1].

Although jailbroken iOS devices allow us to collect more data (e.g., app versions), requiring jailbreaking also would have restricted the size of our userbase, biased our data toward a certain class of users, and prevented us from distributing Carat on the App Store. We opted for less data from more users, and our results demonstrate that energy anomalies diagnosing does not require intrusive instrumentation.

On Android, Carat samples when the `ACTION_BATTERY_CHANGED` Intent fires, at 1% battery level granularity. As we discuss for the remainder of this section, not only is Carat more restricted on iOS than Android with respect to what it can measure, but also when. Carat does not fall into the class of apps that are allowed to run as proper background tasks, which are given intermittent CPU time to perform tasks such as buffering audio, maintaining VoIP server connections, or continuously tracking the GPS coordinates of the device using location services. This means that, in order to take samples while Carat is suspended, our app subscribes to several notifications. When one of these notifications is triggered, iOS allows Carat a small amount of time to take measurements and save these to persistent storage; there is not enough time to communicate with the server.

Carat subscribes to battery-related events (`UIDeviceBatteryLevelDidChangeNotification` and `UIDeviceBatteryStateDidChangeNotification`) and significant location changes (`startMonitoringSignificantLocationChanges`). The location change feature is especially valuable for us. It not only uses far less energy than using the full-fledged location service, but it means that the OS will automatically relaunch Carat if it is terminated while the service is active. (In our deployment, while Carat was in the background, roughly half of samples were triggered by location services and a third were triggered by the battery level event.)

3.1.1 User Interface

When the Carat app is launched, it sends locally stored samples to the server. When Carat is in the foreground, the temporal resolution of sampling increases several-fold.

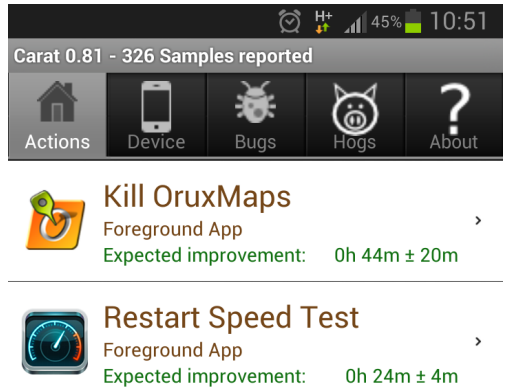


Figure 6. The top of the main screen of Carat on Android, showing recommended actions and projected battery life improvements.

These observations—that increased user engagement leads directly to data being recorded more often and reported sooner—motivated us to spend time honing the user interface, which we now present.

The main screen of Carat is the Actions list, shown in Figure 6, which presents actions the user can take to improve battery life, based on what Carat has learned about their device (e.g., what apps they run), sorted by the expected improvement if that action is taken. For example, the figure shows an action "Kill OruxMaps" that would result in an expected increase of 44m. This means our analysis observed that a typical device running this game will run a full battery down to zero almost 44 minutes sooner than a typical device running typical apps but not OruxMaps. Carat will suggest restarting bugs, admitting the possibility that the instance is caught in a bad state; if restarting does not help, it may be a configuration problem or specific to user behavior. Finally, our current implementation suggests upgrading the operating system if it observes that a newer version is correlated, across the community, with better battery life. The current UI does not reflect all information present in the diagnosis trees; that is planned for a future release.

The Device tab displays information about the client's device, including most of the information that is being recorded and transmitted to our server: the process list, the device model and OS, the state of memory, etc. This tab also prominently displays a number called a J-Score, which is the percentile into which the client's battery life falls within the community; a J-Score of 65 means a better active battery life than 65% of similar devices. Active battery life is computed based on Carat sampling and omits idle periods. This client's average battery drain when using the device would fully deplete the battery in about 16 hours.

We created the J-Score (see Figure 7) to increase user interest and sharing, hoping that it would introduce an element of social competitiveness to energy efficiency. It appears, anecdotally, to have worked. For instance, upon observing that her score had dropped precipitously due to an influx of

Figure 7. The Device tab on the iOS client. The J-Score indicates the percent of the community with worse battery life than this device.

new users, one user remarked (tongue-in-cheek) that she was "no longer confident in our analysis results." She continues to check her score regularly, incidentally sending us samples each time.

The Actions list only suggests killing or restarting an app that is currently active (i.e., in the process list). The Hogs tab shows the top hogs ever reported to have run on the device. The same is true for bugs under the Bugs tab. Clicking on one of the hogs or bugs brings up a detail page where the user can explore the data further.

3.2 Carat Server

The Carat server collects samples from instances of the Carat app running on clients' mobile devices and stores them for use by the backend analysis (see Section 3.3), and it serves actions and other analysis results to clients.

The server is a <1300-line Java application (excluding code auto-generated by Thrift) that listens on TCP port 8080 for incoming client connections. We host with Amazon EC2 because it provides a mechanism to scale the server by spawning new instances and to run a load-balancer to distribute incoming connections.

Received samples undergo lightweight processing to remove junk or malformed data and are then sent to persistent storage. This preprocessing removes OS daemons from the list of processes. We manually maintain a blacklist of such daemons, as it does not appear that the iOS API provides enough information to determine this automatically.

3.3 Backend Analysis

The Carat analysis consists of approximately 5000 lines of Scala, written in the Spark framework [45]. Spark is a cluster computing framework designed for iterative and interactive jobs, distinguished by its use of Resilient Distributed Datasets (RDDs). RDDs are read-only collections of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Parallelism in Spark is provided through operations on the RDDs (e.g., `map`, `reduce`, and `filter`).

Existing data-flow based frameworks such as Hadoop or Dryad depend on intermediate data being written and read from disk, incurring a huge performance hit for iterative jobs. In contrast, Spark provides an efficient environment for multi-stage jobs by reusing the same worker nodes across iterations. In addition, it provides a robust programming model for interactive queries where it is desirable to load data into memory and query it repeatedly (with different filters). These features, along with fault tolerance and its memory management model, made Spark a good fit for implementing Carat's analysis.

The production version of Carat runs in a 20-node cluster composed of high-memory Amazon EC2 instances. This section provides an overview of Spark, the challenges related to parallelizing our analysis, and our solutions.

After converting samples to rates, the computation proceeds in two main stages: identifying hogs and bugs and then generating MCAD trees (see Section 2). The first stage is summarized in Algorithm 3.1

Algorithm 3.1: ANALYZERATES($allRates, aDist$)

```
comment: Hog detection
for each app ∈ allApps
do { filt ← ALLRATES.FILTER(app in _.allApps)
     filtNeq ← ALLRATES.FILTER(app not in _.allApps)
     d' ← COMPAREDISTRIBUTIONS(filt, filtNeq, aDist)
     if d' > 0
       then { comment: store hog and distributions }
comment: Bug detection
for each id ∈ allIds
do { fid ← ALLRATES.FILTER(_.id = id)
     notFid ← ALLRATES.FILTER(_.id!=id)
     comment: Consider apps reported by id, omit hogs
     fidNonHogs ← FID.MAP(_.allApps) \ Hogs
     for each app ∈ fidNonHogs
     do { appFid ← FID.FILTER(app in _.allApps)
          appNotFid ← NOTFID.FILTER(app in _.allApps)
          d' ← COMPAREDISTRIBUTIONS(filt, filtNeq, aDist)
          if d' > 0
            then { comment: store bug and distributions }
     scoreDist ← GETDIST(fid, notFid, aDist)
     comment: Save scoreDist for J-Score calculation
comment: Write J-Scores based on the processed distributions
```

In Section 2.3, we discussed how Carat converts consecutive samples into rates. This computation involves a dependency between samples that complicates the parallelization process.

To remove this inter-sample dependency, we create RDDs of consecutive sample pairs. This new RDD is free of dependencies, so the Spark runtime can independently assign data and conversion tasks to workers. This is done by applying a `map` operation to every item in the RDD. The result of this operation is another RDD consisting of rates. We add metadata for backtracking.

3.3.1 Parallelizing Distribution Building

The bulk of Carat's analysis is the process of building and comparing rate distributions. We load the rates into an RDD, which Spark automatically distributes to all compute nodes. The parallelization strategy must compute distributions on

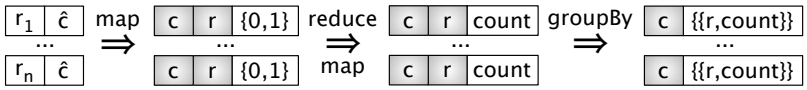


Figure 8. The parallelization process starts with rates as an RDD. Each rate r has features $\hat{c} = (c_1 \ldots c_n)$. To compute rate distributions on feature c (e.g., each app), we map the RDD to a structure with (c,r) as the key (shaded) and, as the value, 1 if the feature occurs and 0 otherwise. A reduce operation yields the rate frequencies for features. We map again, now with c as the key, and (r, count) as the value. Grouping by key then gives the frequency of every R for every F. With slight modifications to the mapping and grouping fields, we use this parallelization strategy for hogs, bugs, J-Scores, etc.

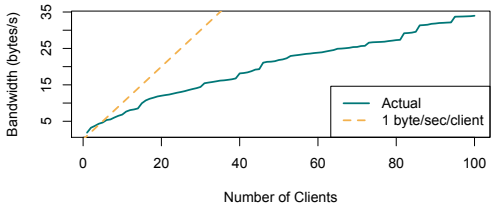


Figure 9. The Carat server sees minimal traffic from individual clients, and the growth of this traffic is linear in the number of users.

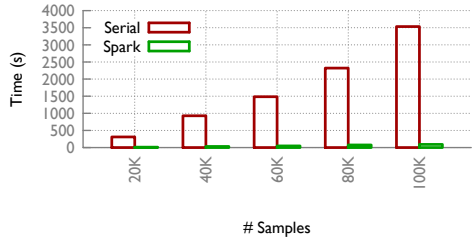


Figure 10. The Carat analysis scales almost linearly when parallelized, while a serial implementation shows exponential complexity.

features in parallel. That is, when building distributions on feature c, the technique must compute distributions for all values of feature c. We devise such a strategy using Spark's RDD operations as follows.

We begin with items in the rate RDD, composed of rates r and their associated features $(c_1, ..., c_n)$, split among worker nodes. We compute distributions of rates conditioned on c and compare them with distributions satisfying $\neg c$. (We compute the distribution for $\neg c$ by subtracting the distribution for c from the full distribution.)

The first step maps items to the format $((c,r),\{0,1\})$, keyed on c and r and with a value of 0 or 1, indicating the presence of the rate, computed from the apriori (see Section 2.2). A `reduce` operation computes the frequency of each (c,r) pair. We remap the reduced RDD and make c the key and (r, count) the value. When we apply a `groupBy` on the key, we obtain the frequency of every rate for every value of c, or a sequence of $(c,(r, \text{count}))$ (see Figure 8).

We now have two RDDs, one with the frequency of rates satisfying c and its complement. The RDDs are joined using a `groupWith` operation. A final `map` operation passes them through our distribution building and comparison module in a parallel fashion, thus obtaining the expected improvements and the correlations. The same parallelization strategy is applied to compute hogs (features are apps), bugs (features are (UserID, App) pairs), J-scores (features are UserIDs). We observe that most other feature-grouping required in Carat's analysis can be reduced to this parallel model.

3.4 Performance and Scaling

The success of our approach depends on an active community and generates better results as that community grows, so the implementation must be scalable.

Our frontend experienced linear traffic scaling with the size of our deployment, at a rate far below 1 byte per second per client (see Figure 9). Sample reporting is presumed to be unreliable; a client with no disk space or network access is allowed to throw away samples and an overloaded server may drop packets. Five medium Amazon EC2 instances behind an Elastic Load Balancer (ELB) has been handling our userbase of half a million devices.

Our current implementation of the analysis backend (see Section 3.3) uses the Spark cluster computing framework. The computation is massively parallel, as every distribution and comparison can be computed independently. Figure 10 compares the runtime for an optimized serial implementation of the analysis algorithm compared to a parallel implementation in Spark for increasing number of samples. The results underline the need for parallelization. As our userbase grew, we made numerous optimizations. The analysis program now computes all reports for all our users (24 million samples) from scratch in approximately 45 minutes.

4 Ground Truth and Overhead

For Carat to accurately account for when energy is being used, it must convert intermittent (low precision) battery level samples into energy drain rates in a way that is faithful to the ground truth. Furthermore, the practicality of our method relies on sampling that is sufficiently low-overhead that it does not have a significant impact on the energy use, itself. In this section, we attach mobile devices to power metering hardware: an iPhone 4S to a Monsoon Power Monitor[2] (see Figure 11) and a Galaxy Tab 2 10.1 to Leyden Energy's[3] battery-testing equipment. Our results confirm that Carat generates accurate energy distributions while consuming few resources (i.e., almost no battery).

To test the fidelity and cost of our sampling, we ran the devices through a script of varied activities. The script is not intended to be a representative workload, but to repeatably exercise the device features and drain the battery at different

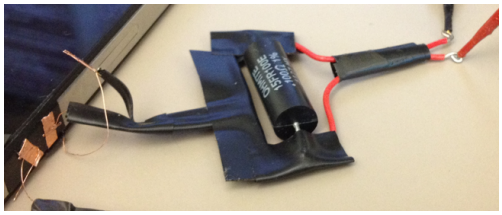


Figure 11. Close-up of the wiring rig that connects our iPhone 4S test phone with the Monsoon Power Monitor.

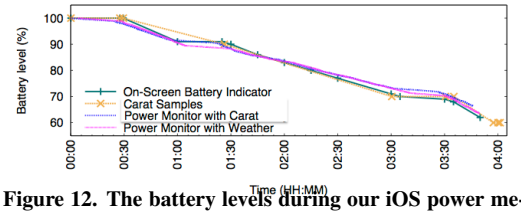


Figure 12. The battery levels during our iOS power metering experiments, either taken directly from the on-screen battery indicator, the Carat samples, or computed from the meter's readings.

rates. It includes such behaviors as downloading and running an app, browsing the web, playing a game, and idle periods. The WiFi was turned on for some periods and off for others.

On each device, we ran through the script under three different arrangements: (1) hooked up to the power meter with and (2) without Carat running and (3) not hooked up to the power meter with Carat running. We compare the data from (1) and (2) to quantify the overhead of running Carat; we compare the data from (1) and (3) to ensure the meter was not influencing Carat's measurements and to assess the fidelity of our sampling and rate estimation. For the runs performed without Carat, where our app appears in the script, we substituted the standard Weather app.

The battery levels reported by the OS, both through the API (Carat samples) and the on-screen indicator, track the actual use of power by the device. Figure 12 shows the iOS data. Between 00:30 and 1:30, Carat took no samples and conflated a higher-rate period with a lower-rate period. Higher frequency sampling would have avoided this error.

The expected energy discharge rates computed from the Carat samples approximate the values computed using power metering hardware. During the 9-hour iOS experiment, Carat took 9 samples at 5% granularity; the power meter took 13,549 samples at effectively 0.0001% resolution. Carat overestimates the average discharge rate by only 0.00088%/sec (see Figure 13). On the Galaxy Tab, where Carat took twice as many samples as on iOS (19), the error is an order of magnitude less (0.00015%/sec). This accuracy is possible thanks to the *a priori* distribution, which uses knowledge of community behavior to refine noisy and incomplete measurements; imprecision in per-client measurements is further mitigated by the statistical backend analysis.

Carat imposes negligible energy overhead. Our power metering hardware indicates that running through our iOS script with Carat running used *less* energy (53.691 mAh or

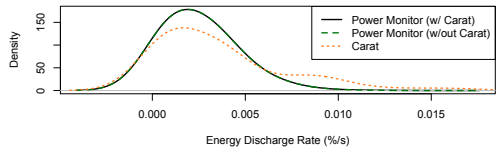


Figure 13. The energy rate distributions from our iOS power metering experiments, smoothed with a Gaussian kernel estimator for visibility. Using the *a priori*, Carat is able to faithfully estimate the distribution with sparse sampling, overestimating the mean energy drain rate by only 0.00088% from 9 samples.

∼3.5% of the battery less) than executing that same script with the Weather app running in its place (i.e., 54 minutes less battery life running Weather instead of Carat). We also ran the script without substituting another app but found battery life with Carat running was slightly higher than without; Carat's energy use is less than the experimental precision. Similar results held on Android. We can afford to perform sparse, low-overhead sampling on individual clients because we aggregate such data from many clients.

5 Deployment Evaluation

Carat became available as a free download on Apple's App Store and on Google's Play Store in mid-June of 2012. Days later, it was featured on the popular TechCrunch blog[4]; the story was soon picked up by dozens of other news sources. Within 24 hours of the article's publication, we went from a few hundred users to more than 100,000. This doubled in the subsequent 24 hours. Carat has been installed more than 560,000 times; of those, 475,041 clients reported data (some never ran the app or never when connected to the internet); 409,867 reported enough data to yield diagnoses.

Our salient results (see Sections 5.4–5.7) are that we found no instances of false positives among the reported anomalies; after two weeks, users who received Carat recommendations improved battery life by 13% (c.f. 3% for those who did not); and 95.2% of the predicted battery life improvements fell within the predicted 95% confidence bounds.

5.1 Data

Our users ran iOS (55%) and Android (45%). Tables 1 and 2 show breakdowns of the most common device models and operating systems. In aggregate, the devices recorded 16.5 million rates, launching our app 7.4 million times (a median of 1.9 sessions per day).

The community ran 102,421 different apps, with a disproportionate number (56%) coming from Android users. Of these apps, 10,110 (9.9%) were classified as hogs, of which 83% were Android apps. Carat detected energy bugs in thousands of apps; of the 21,529,249 total possible bugs (user-app instance pairs), 1.1% were classified as such.

Clients reported samples at a wide variety of rates, clustering into casual users recording a few samples daily and heavier users sampling sometimes a hundred times as often. The average number of samples per day was nearly the same on both platforms (36.8 samples per user per day on iOS and 37.7 on Android), but the variance of this rate on Android

Device Model	Number	% Total	% Platform
iOS			
iPhone 4S	85,267	20.8	37.6
iPhone 4	54,853	13.4	24.2
iPhone 5,2	12,590	3.07	5.56
iPhone 3GS	12,364	3.02	5.46
iPhone 5,1	12,239	2.99	5.40
Other	49,258	12.0	21.7
Android			
unknown	22,057	5.38	12.0
GT-I9100	15,770	3.85	8.60
Galaxy Nexus	10,333	2.52	5.64
GT-I9300	7238	1.77	3.95
GT-N7000	5009	1.22	2.73
Other	122,889	30.0	67.0

Table 1. The most common device models in our deployment, showing the percent of users from whom we had sufficient data to generate diagnoses.

OS Version	Number	% Total	% Platform
iOS			
5.1.1	136,485	33.3	60.2
6.0	35,708	8.71	15.8
6.0.1	21,068	5.14	9.30
6.1	10,009	2.44	4.42
Other	23,301	5.69	10.3
Android			
4.0.4	40,512	9.88	22.1
4.0.3	24,439	5.96	13.3
2.3.6	19,782	4.83	10.8
unknown	18,075	4.41	9.86
Other	80,488	19.6	43.9

Table 2. The most common operating system versions in our deployment, showing the percent of users from whom we had sufficient data to generate diagnoses.

was 32% higher than on iOS. This is, in part, because some Motorola devices only triggered the battery level intent at 10% levels while most other Android devices triggered every 1%; iOS devices triggered consistently at 5% increments.

5.2 User Behavior

The frequency and duration of user engagement matters. The more often users launch Carat, the fresher our data will be (that is when it is sent to our server). On both iOS and Android, the longer users keep Carat in the foreground, the more samples it can record. The session length data (see Table 3) and click-path data show that many stay in the app to explore the reports or check their J-Score. Almost half of the sessions last more than 30 seconds.

Figure 14 shows the period of time over which users open Carat. After a month, we retain roughly 25% of our users; only 6% use the app for more than 90 days. The median user opens Carat 1.9 times per day and 3.0 times per week.

5.3 Injected Anomalies

We added energy anomalies to an existing app—initially with no apparent misbehavior—to confirm that Carat is able to detect the new bugs. We chose the Wikipedia Mobile app made by Wikimedia Foundation for iOS because it is an open-source app used by many of our clients but was not reported as an anomaly. We added several behaviors to the Wikipedia app that consume large amounts of energy when activated, with each one repeatedly using a different resource: radio, CPU, and GPS.

Session Length	Sessions	% of Sessions
0–3 secs	257,632	4.15
3–10 secs	893,793	14.4
10–30 secs	2,100,538	33.9
30–60 secs	1,397,873	22.5
1–3 mins	1,109,035	17.9
3–10 mins	163,478	2.63
10+ mins	282,645	4.56

Table 3. The length of Carat sessions. The app only reports data when it is opened and can sample more aggressively in the foreground. So, incentivizing the user to open the app and explore results from within the UI helps us collect more data.

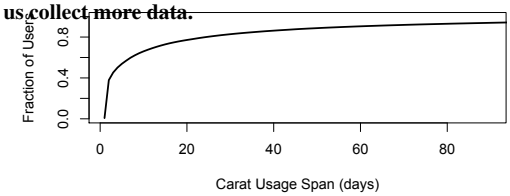


Figure 14. The number of days over which Carat users open the app. Some users check Carat only over a period of several days (to see their initial reports) and then never again; the majority, however, check back with the app occasionally over the following weeks or months.

We installed the buggy Wikipedia instance on one of our test devices, an iPhone 3GS. Wikipedia Mobile was already in use by several clients at this point, so a baseline distribution had been established and Carat did not consider the app to be anomalous. We ran the app for one day for each injected bug (i.e., radio, CPU, and GPS), activating the app a handful of times during the day but only leaving it open for a couple of minutes (casual use). At the end of the third day, we ran the analysis with the real, non-buggy data as the reference distribution and once each with the data from exactly one of the buggy days as the subject distribution. Thus, we could declare success if the analysis reported three bugs, one for each injected behavior.

Indeed, after performing the injection, Carat correctly detected each of the three bugs (no false negatives). Figure 15 shows the reference distribution and each of the three subject distributions for the iPhone 3GS running our buggy Wikipedia build. The expected improvement reported for fixing each bug (i.e., returning the app to typical Wikipedia Mobile behavior) was 27m 26s for the CPU bug, 9m 22s for the GPS bug, and 55m 28s for the Radio bug, which agreed with what the experimenter observed on the device.

5.4 Wild Anomalies

Carat detected 10,110 hogs and 233,258 buggy app instances among the 102,421 apps run by the 409,867 users for whom we had sufficient data to generate reports. We ranked the hogs and bugs by a function of severity (predicted battery impact) and popularity (number of users than ran the hog or had a buggy instance), resulting in one list for each kind of anomaly. Although our manual validation process prevented us from checking the entire list, we did check the first two dozen from each list using a combination of user complaints, news coverage, analysis tools (see Section 5.4.1), or experi-

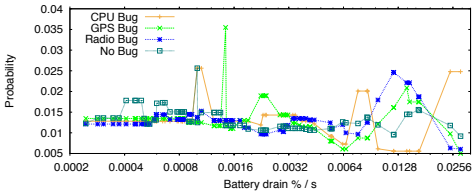


Figure 15. The reference (anomaly-free) and anomalous rate distributions for the modified Wikipedia Mobile, using only the a priori from the private deployment. Carat successfully detects all of the injected bugs.

mental results in the literature (e.g., [32, 33]). Among these anomalies, there were no false positives. Later in this section, we describe a subset of these manually-checked anomalies that we feel highlight interesting circumstances or salient aspects of our analysis (see Sections 5.4.2–5.4.3). Note that the number of apps for which we performed manual validation (∼50) already makes this paper a high-water mark for evaluating energy diagnosis on mobile devices, even without considering the other 100,00+ apps that Carat analyzed or the many thousands of diagnoses it generated.

Our attempts to acquire the tools used in prior work to validate our results were unsuccessful; the authors either did not respond, told us the tools were not in a state to be used by people other than themselves and they didn't have time to help us, or they simply refused to furnish the tool. Regardless, no existing tool that we know of would have allowed us to validate all tens of thousands of anomalous apps and app instances that Carat discovered.

5.4.1 External Validation with ARO

AT&T provides a tool called the Application Resource Optimizer (ARO) that uses network traces to identify communication-related misbehavior. We selected the four most severe hogs (GO SMS Pro, Advanced Task Killer, Line: free calls and messages, and Chant for Twitter) and four non-anomalies (Lookout Antivirus, Facebook, Gachinko Tennis, and Dropbox) on Android that showed a strong correlation between increased energy use and network connectivity.

The tool indicated that all four hogs had bursts of network communication that could be more tightly grouped. Three were missing cache headers that might have reduced retransmission; the fourth, Advanced Task Killer, was implicated for wasting energy by not closing network connections. Although half the non-anomalies also lacked cache headers, they did not perform redundant downloads like some of the hogs. ARO corroborated these hogs, but also gave some indications of misbehavior by the non-anomalies; only the accompanying energy measurements separated the misbehavior that hurts battery life from that which doesn't. Furthermore, without a collaborative method like Carat that collects data from multiple devices, it is hard to say whether any of this behavior is intrinsic to the app or a function of device- or user-specific factors.

5.4.2 Hogs

Of the 102,421 apps seen during our deployment, 10,110 (9.9%) were categorized as hogs. (Before checking for statistical significance, there were 15,038 (14.7%).) Recall that an app is a hog if the community-wide average discharge rate while running the app is significantly greater than the average rate while not running it (see Section 2.1) and that we can compute the expected improvement in battery life by killing a hog (see Section 2.4). Hogs may be caused by an oft-triggered code bug or may be simply intrinsic to the app. Users concerned about battery life are advised by the Action list to kill hogs; the user is not concerned about the intention, or lack thereof, behind the energy use.

While some hogs were unsurprising to us (e.g., Pandora and Skype), others were (e.g., some Android themes and wallpapers). For instance, while most apps for searching airline fares and booking flights are not among the hogs—they use the network but not heavily and do not use many other resources—there were a handful of such apps that appeared among the top hogs. We discovered that all those airline apps were written by the same developer and were suffering from a systematic programming inefficiency.

The top ten hogs (by severity) on iOS all fall into the category of utilities, including iDesp Money (for budget management), Ushahidi (for sharing stories within a community), and the Citi Mobile banking app. There were no games; despite being typically resource-intensive, they did not use energy as anomalously as other kinds of apps. Similarly, the top hogs on Android were primarily utilities, but there were also several wallpaper apps (e.g., Beach at Night and Heart and Love) and one game (which has since been removed from the app store).

We now describe a couple of hogs from among those we manually checked (again, there were no false positives) and cite corroborating evidence that the app does, indeed, consume an anomalously large amount of energy.

Pandora Radio: Carat classifies Pandora Radio, which 7116 iOS users ran, as a hog and says killing it will increase an client's average battery life by 50m 43s. This is corroborated by user reports, one of which claimed Pandora drained the battery to 30% in a few hours even with the screen off[5]. To improve battery life while using Pandora, the MCAD suggests using WiFi for connectivity (an additional 25–35m). Pandora is an example of an intuitive hog, as it uses several energy-hungry resources, but Carat quantifies the cost.

Skype: 27,741 iOS clients were running the Skype VoIP app, which was also reported as a hog. This is also confirmed by the forums; one user even used the term "power hog" to describe Skype[6]. Skype's energy use is driven by network connectivity; when no network connection is available, expected battery life is about 6.5h above average.

Go launcher exe new theme. . . : (*sic*) Is an unlikely hog on the Android platform that costs most users between 2h 1m and 2h 53m of battery life. Experiences with Go Launcher and its variants, which change the UI of the device, vary among users[7], but generally "fancier" themes and widgets cause higher battery drain[8].

Live wallpapers: Carat identifies several Android Live

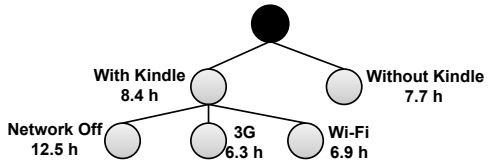


Figure 16. MCAD for the Kindle app on iOS, showing the expected battery life when using exclusively this app under various conditions. The diagnosis points to network connectivity as the primary determinant of energy use. Note that, as with all bugs, Kindle uses less energy than a typical app ("Without Kindle") when the bug is not triggered.

Wallpapers as energy hogs. Two that rank among the top 10 most severe hogs on the Android platform are Beach at Night[9] and Heart and Love[10]. They cost most users 2h 33m–2h 49m and 2h 37m–2h 51m battery life, respectively. Both are ad-supported; the detrimental effects of adware are known [32]. Both live[11] wallpapers[12] and adware[13] have been blamed for abnormally fast battery drain.

5.4.3 Bugs

Recall that a bug is an app that is not a hog (it usually consumes below-average energy) but consumes far more energy on some clients than others (see Section 2.1). Although the current Carat client-side UI only suggests restarting a bug (in case it is simply caught in a bad state), the MCAD diagnosis computed on the backend enables more specific recommendations, such as disabling WiFi or turning on GPS; we plan to add this in later versions of the app. Note that, without a community of clients, distinguishing bugs from hogs would be impossible and identifying the triggers would be difficult.

The maximum number of bugs that Carat could report is the sum over clients of the number of non-hog apps they ran, which was 9.1 million in our dataset. Our method reported 233,258 buggy app instances (1.1%); we describe some examples below.

Many popular apps, including Facebook and Youtube (on iOS) and Twitter and Chrome (on Android), exhibit anomalously high energy use among small subsets of users. This suggests that those apps have configurations or usage modes that consume significantly more energy. By severity, however, most of the bugs are again less popular utilities: e.g., Koder and Raved on iOS and Police Scanner and Are You Watching This?! on Android. There were two games among the top ten most severe bugs: Tower of Fortune (iOS) and Papaya Diamond (Android). Unlike the Android hogs, no wallpapers were among the top bugs.

Kindle: This electronic book app was reported as a bug for 254 out of 2617 iOS clients (9.7%). Figure 16 shows a diagnosis tree for Kindle, in which 3G connectivity appears especially detrimental. The support forums blame the problem on WhisperSync[14], which synchronizes notes, bookmarks, previous location, and Popular Highlights. When syncing over GSM, in particular, the device uses much more energy than syncing over WiFi. Our data support this hypothesis, which had previously been only anecdotal.

Facebook Messenger: Was anomalous on 792 of 7350 Android clients (10.8%). The MCAD indicates that upgrading the OS improves battery life (71–83m), and that WiFi is more energy efficient than other connectivity options. Using the app while stationary gives a 63–97m boost to battery life. (Note that Carat does not advise users to stand still.)

YouTube: Was a bug on 3118 of 37475 iOS clients (8.3%). The MCAD shows that while moving, users of mobile Internet have a battery life advantage over WiFi users (25–34m). When compared to immobile WiFi users, mobile network users still have a 20–28m advantage. This is contrary to many apps, where WiFi is less energy-consuming.

Twitter: Was reported as a bug on 2744 of 18651 Android clients (14.9%). The MCAD for Twitter indicates that the most critical cause of battery drain is an old OS version. Users of Ice Cream Sandwich (4.0.4) got 94m to 100m more battery life than other Android Twitter users. Use of WiFi with 4.0.4 yielded another 85m to 105m; this was not observed on other OS versions.

SwiftKey: A popular keyboard application for Android, SwiftKey is one of the top 15 bugs by severity, affecting 2402 users. The developer website indicates that the latest release of the app exhibits high energy drain, especially in newer versions of Android OS[15].

5.5 Diagnosis on Other Features

Carat analyzes the battery life implications of many other combinations of features on the backend as part of the MCAD generation, including the OS version, device model, internet connectivity, and so on. For various reasons, the Carat UI does not recommend that a user take actions like purchasing a different device model or downgrading to an earlier operating system version (those features are not actionable, as discussed in Section 2.6). Other than killing or restarting apps, the only action our current Carat implementation might suggest to users is to upgrade the operating system.

iOS 5.0.1: Shortly after Apple released iOS 5.0, many users complained of issues with poor battery life. The subsequent point release—iOS 5.0.1—was touted, in part, as a fix for these problems. The public reaction was mixed[16]. One user said, "After updating I am seeing my power drain at a much quicker rate"; another claimed his phone was "Still draining at the exact same rate"; and a third, meanwhile, reported that his battery life was "doing much better." In summary, users had a wide variety of anecdotes but no data.

Using the data from our deployment, Carat discovered that, in fact, the average discharge rate for devices running 5.0 was higher than for devices running 5.0.1. Clients running 5.0.1 should expect to see, on average, a 1h 11m 30s increase in battery life, supporting Apple's claims that the update addressed some of the battery problems in the initial release. Users running iOS 5.0 at the time 5.0.1 was released (and this diagnosis was computed) were advised by our app to upgrade.

5.6 Battery Life Improvement

One key metric metric is whether battery life tends to increase over time for our users, a coarse measure of whether using Carat reduces energy use. The metric is coarse be-

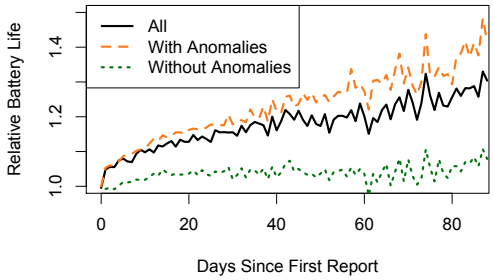


Figure 17. Average relative battery life of Carat users following the generation of their first report (hog and bug lists), using the battery life of the first day as the baseline. A typical user (black line) sees an 7.0% increase after a week, surpassing 23% after two months.

cause it includes several confounding factors: some of these users may not have followed Carat's recommendations, the population is biased toward users who originally had battery problems (and thus installed Carat), and users may have also employed alternative means to decrease energy use. Some users did not run any apps that Carat considers anomalies and therefore did not receive any reports; that is our control group. Figure 17 shows average relative battery life over time for Carat users who did ("With Anomalies") and did not ("Without Anomalies") receive reports. (The increased variance at higher "Days Since First Report" is due to user attrition; see Figure 14.)

After 2 weeks, the average user sees an 11.7% improvement in battery life, however, users who received reports saw a 13% increase while those who did not gained only 3%. This is more pronounced for long-term users (90+ days); when Carat recommended battery-saving actions, users improved battery life by 41%, compared with 7.9% when Carat did not.

Although users who received recommendations from Carat had a marked improvement in battery life, we considered the possibility that the improvement may have arisen through actions other than those specifically suggested by our app. For example, upon being told to kill App X, the user might instead simply restart their phone, kill all the running apps, or coincidentally stop using App X as part of normal app turnover. This may be partly true, but the data also clearly show that users are performing the actions Carat presents to them; after receiving their first report, anomalous app usage (hogs and bugs) decreased by 60%. This is almost double the decrease for non-anomalous apps (33%). (A number which is probably higher than turnover in the general population due to the more prevalent device restarting and app killing among our users.)

These data suggest that not only do users who receive reports manage to significantly improve their battery life, but that they are following the recommendations contained in those reports. Performing the Carat actions yields increased battery life.

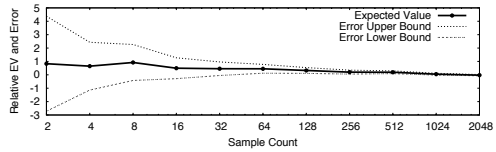


Figure 18. As the number of samples increases, the relative error in our estimate of the expected discharge rate shrinks rapidly. Above is the average expected value for several of the largest anomalies seen in our deployment and the 95% confidence error envelope.

5.7 Improvement Prediction Accuracy

A second key metric is how closely the Carat Actions—and the projected benefits—match the observed benefits. Specifically, when Carat predicts that killing/restarting an app a will improve battery life by $b \pm e$ seconds with 95% confidence, how often is it correct? We found that Carat tended to underestimate the improvement that clients would experience, but 95.2% of these predictions fell within our 95% confidence bounds.

We reached this number using the following analysis. Let $x_{u,a}$ be the fraction of the time that user u reports running app a, within some window of time. The estimated battery life improvement b (in seconds) that Carat quotes assumes a transition from $x_{u,a} = 1 \rightsquigarrow 0$. We assume that the achieved benefit is linear in Δx, so moving from $x_{u,a} = 1 \rightsquigarrow 0.5$ (using the app half half the time instead of all the time) yields an improvement of $0.5b$ seconds; transitioning from $x_{u,a} = 0.5 \rightsquigarrow 0.3$ yields an improvement of $0.2b$ seconds. (Other actions that Carat suggests, such as upgrading the operating system, cannot be done fractionally.) The predicted benefit b is therefore a slope; we compare the predicted improvement curve $y = bx$ (and error margins) with the empirical curve—a least-squares best-fit line through the actual battery life and usage numbers collected by the app—with slope b'.

As stated above, the data show that if Carat advises killing an app and that doing so will increase battery life by $b \pm e$, then across all recommendations made by Carat there is a greater than 95% chance that decreasing the frequency of app use will result in the projected improvements (subject to the scaling described above).

As the number of clients and samples increases, so does the accuracy of our predictions. In particular, Carat's estimate of the expected value—the crucial number used to identify anomalies and compute expected benefits—tends to converge to the true value. Figure 18 shows the shrinking relative error envelope of this estimate for some of the anomalies Carat detected in the wild.

There is no guarantee of convergence in practice because the true rate distribution may be neither stationary nor identically distributed. Indeed, this paper has discussed at length one situation where a rate distribution may not be identically distributed across clients: the presence of an energy bug. As long as a bug affects a constant fraction of the population, however, this convergence happens almost surely, in the mathematical sense (as the number of samples goes to infinity, the estimated expected value converges to the true

value with probability 1).

6 Limitations and Future Work

Carat takes a black-box approach to diagnosing anomalies, which carries inherent limitations. Without visibility into the mechanisms (e.g., code, messages, or kernel state) and without the ability to perturb the system (i.e., it is passive and cannot modify other apps), the best possible result is to say what aspects of the system are likely to be involved with the abnormal battery discharge. This is what Carat provides, and it does so by correlating real-valued signals from features without initial assumptions about their relationships. This kind of approach has proven fruitful in prior work [26, 28].

Compared to iOS, Android provides greater visibility into the behavior of apps and the operating system, as would facilitating app instrumentation through a developer API. We opted for feature parity with iOS for this paper in order to evaluate a method that works for both platforms, but plan to leverage such additional data in later versions of the app (and already do so on the backend).

As with any passive approach, which a regulation iOS app must be, our results are limited by the data. If none of the clients ever runs a particular buggy app, Carat will never detect a problem; if two apps are always run together and one is anomalous, they will both be categorized as anomalies and there is nothing that correlation can do to disambiguate. The likelihood of spurious correlations increases with the number of features (apps and configurations). The way to combat this problem is with more data. For example, as we gather more samples involving highly correlated apps that show one but not the other, we can begin to discern which (or possibly both) are responsible for the anomaly. The results show that our data are sufficient for actionable diagnosis.

Carat is targeted at users, but additional in-app instrumentation (such as via a developer API) would enable finer-grained diagnoses for developers, e.g., identifying what user behaviors, app settings, or other environmental conditions trigger abnormal energy use.

7 Related Work

There is a rich body of work in diagnosis for correctness and performance. Recent work identified an emerging class of software misbehavior that afflicts battery life [31] and proposed a method for detecting a specific class of such bugs [33]. We believe our work is the first collaborative method to automatically detect and diagnose abnormal energy use on mobile devices. Unlike previous work, Carat is able to disambiguate between hogs and bugs—anomalies that are intrinsic to an app versus those that may be triggered by device- or user-specific conditions, respectively—a capability that requires measurements from multiple devices. An early prototype and small deployment of the method on a single platform was summarized in our workshop paper [27].

Our approach is a form of statistical debugging, in which (loosely speaking) deviant behavior is called a bug [9]. Such methods have been used to identify code paths correlated with failure [16, 17], concurrency bugs [14], shared influence (surprising behavior that is correlated in time) [26, 28], invariant violation [13], and configuration errors [41]. In the field of security, anomaly-based intrusion detection has a long history [8, 34, 35]. Recently, statistical methods were used to diagnose energy problems by comparing the behavior of an app at different times on a single device [21]; this kind of approach cannot disambiguate hogs from bugs or separate app-intrinsic behavior (many apps consume different amounts of energy depending on what features are being exercised) from device- or user-specific factors.

These statistical methods frequently make use of a large number of instances or users of these programs, which is sometimes called a *community*. A recent paper suggests a collaborative debugging framework called MobiBug for mobile devices [1], but they focus on crashes, not continuous or intermittent measurements. There is prior work for file systems [42] and peer-to-peer networks [22] that generate alerts based on aggregate behavior.

Projects like the Application Communities project [20] use the community to distribute work; instead, we employ uniform, lightweight instrumentation. There are also security applications for the community besides detection, such as diagnosing problems by discovering root causes [41] and preventing known exploits (e.g., sharing antibodies) [7, 25].

Many projects have sought to profile or emulate energy use on mobile devices [10, 11, 23, 29, 30, 32, 44], sometimes for prediction [37, 40], mitigation [3, 18], diagnosis [21], or developer tools [15]. Human interface studies have shown that 80% of mobile users will take steps to improve their battery life [36]; Carat recommends specific, personalized actions for users to take and even estimates the benefit they are likely to see. This is a distinguishing feature of our work.

Energy debugging shares similarities with performance debugging; both areas aim to account for the use or abuse of a shared resource. Some notable performance debugging work includes history-based analysis in datacenters [5], resource accounting [4], and blackbox debugging [2]

Pinpoint [6] and Magpie [4] track communication dependencies with the aim of isolating the root cause of misbehavior; they require instrumentation of the application to tag client requests. In order to determine the causal relationships among messages, Project5 [2] and WAP5 [38] use message traces and compute dependency paths. D^3S [19] uses binary instrumentation to perform online predicate checks. Recent work shows how access to source code can facilitate tasks like log analysis [43] and distributed diagnosis [12]. CarrierIQ[17] collects detailed measurements by integrating with the mobile platform, and has drawn criticism for the intrusiveness of their implementation[18]. Unlike the preceding methods, we do not assume such access to code, communications, or binaries, taking instead a black-box approach with broader deployment potential.

8 Conclusions

This paper presents a method for diagnosing energy anomalies in the wild given incomplete and noisy instrumentation measurements from a community of clients. We implemented this method as an app for iOS and Android called Carat and deployed it to a community of more than

500,000 devices. Carat diagnosed thousands of anomalies, which involves detecting the anomaly, estimating its severity, quantifying the error and confidence bounds on that estimate, and sometimes identifying the device features that are correlated with the anomaly. We also validated our implementation with hardware measurements and synthetic anomaly injection, showing that Carat can accurately estimate energy use and detect anomalies.

Specifically, Carat imposes negligible overhead on each device, estimates energy use with accuracy comparable to hardware, detected 100% of synthetically injected anomalies in controlled experiments, produced no known false positives (based on corroborating dozens of anomalies using other methods), and predicted the battery impact of anomalies with greater than 95% accuracy. Finally, users receiving reports from Carat improved their battery life by 21% after a month; users who received no reports gained only 5.5% over the same period.

A collaborative approach is required to diagnose energy bugs; even complete knowledge of app behavior on a single client could be specific to a device or user. We believe this is the first collaborative diagnosis of energy anomalies in the wild and represents a crucial extension of previous work in distributed and statistical debugging to include a new class of abnormal behavior related to mobile energy use.

Notes

[1] http://carat.cs.berkeley.edu
[2] http://msoon.com/LabEquipment/PowerMonitor/
[3] http://www.leydenenergy.com/
[4] http://techcrunch.com/2012/06/14/carat-battery/
[5] http://bit.ly/yTIUeU
[6] http://bit.ly/wsMraK
[7] http://bit.ly/WZ4dQi
[8] http://bit.ly/QSiv72
[9] com.bobisoft.wallpaper.beachatnight
[10] com.custom.lwp.FREE_HeartAndLove
[11] http://bit.ly/QSixvT
[12] http://bit.ly/TLWRhV
[13] http://bit.ly/Scgjs2
[14] http://gdg.to/xeK9CZ
[15] http://bit.ly/ODNyxQ
[16] http://zd.net/y0dyCr
[17] http://www.carrieriq.com/
[18] http://onforb.es/zdlzmF

Acknowledgements

We thank the folks at Monsoon for helping us connect their hardware to the iPhone, the beta testers and /r/compsci community of reddit for trying early versions of the app, and our colleagues in the AMP Lab for their valuable feedback. We are also grateful to our numerous reviewers for their comments and to our shepherd, Geoffrey Challen, for helping us improve the paper.

This research is supported in part by NSF CISE Expeditions award CCF-1139158, gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

9 References

[1] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. There's an app for that, but it doesn't work. Diagnosing mobile applications in the wild. In *HotNets*, 2010.
[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Methitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.
[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *IMC*, 2009.
[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.
[5] P. Bodik, M. Goldszmidt, A. Fox, D. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Eurosys*, 2010.
[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *DSN*, June 2002.
[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, 2005.
[8] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium on Security and Privacy*, 1992.
[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, 2001.
[10] D. Ferreira, A. K. Dey, and V. Kostakos. Understanding human-smartphone concerns: A study of battery life. In *Pervasive*, 2011.
[11] J. Flinn and M. Satyanarayanan. PowerScope: a tool for profiling the energy usage of mobile applications. In *WMCSA*, 1999.
[12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.
[13] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.
[14] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.
[15] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. In *HotMetrics*, 2008.
[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
[18] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. Energy-accuracy trade-off for continuous mobile device location. In *MobiSys*, 2010.
[19] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: debugging deployed distributed systems. In *NSDI*, 2008.
[20] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2005.
[21] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI*, 2013.
[22] D. J. Malan and M. D. Smith. Host-based detection of worms through peer-to-peer cooperation. In *ACM Workshop on Rapid Malcode*, 2005.
[23] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Mobicom*, 2012.
[24] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 1997.
[25] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.
[26] A. J. Oliner and A. Aiken. Online detection of multi-component interactions in production systems. In *DSN*, 2011.
[27] A. J. Oliner, A. P. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica. Collaborative energy debugging for mobile devices. In *HotDep*, 2012.

[28] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *DSN*, 2010.

[29] E. Oliver. The challenges in large-scale smartphone user studies. In *HotPlanet*, 2010.

[30] E. A. Oliver and S. Keshav. An empirical approach to smartphone energy level prediction. In *UbiComp*, 2011.

[31] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *HotNets*, 2011.

[32] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with eprof. In *EuroSys*, 2012.

[33] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Mobisys*, 2012.

[34] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Computer Networks*, December 1999.

[35] P. A. Porras and P. G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. In *NIST/NCSC*, 1997.

[36] A. Rahmati, A. Qian, and L. Zhong. Understanding human-battery interaction on mobile phones. In *MobileHCI*, 2007.

[37] N. Ravi, J. Scott, L. Han, and L. Iftode. Context-aware battery management for mobile phones. In *PerCom*, 2008.

[38] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

[39] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers—a survey. *Trans. on Sys., Man, and Cybernetics*, 2005.

[40] A. Sinha and A. P. Chandrakasan. JouleTrack: a web based tool for software energy profiling. In *DAC*, 2001.

[41] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[42] Y. Xie, H. Kim, D. O'Hallaron, M. Reiter, and H. Zhang. Seurat: a pointillist approach to anomaly detection. In *RAID*, 2004.

[43] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[44] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *USENIX Technical*, 2012.

[45] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

Research Theme C: Collaborative Energy Awareness

Research Paper IV

Kumaripaba Athukorala, Antti Jylhä, Eemil Lagerspetz, Maria von Kügelgen, Adam J. Oliner, Sasu Tarkoma, Giulio Jacucci

How Carat Affects User Behavior: Implications for Mobile Battery Awareness Applications

In *Proceedings of CHI Conference on Human Factors in Computing Systems (CHI'14)*, ACM, 2014, pp. 1029-1038

Contribution: This work was directed by K. Athukorala. This publication used data gathered by the Carat Android application, and questionnaire responses from its users. M. von Kügelgen designed the questionnaire together with K. Athukorala. The author designed, implemented, and tested the questionnaire module for Carat that was used to gather questionnaire results from Carat users. The author created the method to match Carat user data with questionnaire responses, and analyze how often users stopped using Hogs and Bugs. The questionnaire responses were analyzed by Athukorala and von Kügelgen.

How Carat Affects User Behavior: Implications for Mobile Battery Awareness Applications

Kumaripaba Athukorala [‡§] **Eemil Lagerspetz** [‡§] **Maria von Kügelgen** [‡§]
Antti Jylhä [‡§] **Adam J. Oliner** [†] **Sasu Tarkoma** [‡§]
Giulio Jacucci [¶§]

[‡] Helsinki Institute for Information Technology HIIT, University of Helsinki, [†] UC Berkeley
[§] Department of Computer Science, University of Helsinki
[¶] Helsinki Institute for Information Technology HIIT, Aalto University
§ first.last@helsinki.fi, † oliner@eecs.berkeley.edu

ABSTRACT
Mobile devices have limited battery life, and numerous battery management applications are available that aim to improve it. This paper examines a large-scale mobile battery awareness application, called Carat, to see how it changes user behavior with long-term use. We conducted a survey of current Carat Android users and analyzed their interaction logs. The results show that long-term Carat users save more battery, charge their devices less often, learn to manage their battery with less help from Carat, have a better understanding of how Carat works, and may enjoy competing against other users. Based on these findings, we propose a set of guidelines for mobile battery awareness applications: battery awareness applications should make the reasoning behind their recommendations understandable to the user, be tailored to retain long-term users, take the audience into account when formulating feedback, and distinguish third-party and system applications.

Author Keywords
user retention; user behavior; smartphone; energy awareness

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous; H.1.2. User/Machine Systems: Human Factors

INTRODUCTION

Mobile devices have limited battery life, sometimes requiring a recharge more than once per day. Rapid energy drain may be caused by extensive use of resources (e.g., the network, CPU, or GPS) by running applications or the device operating system, itself. Poor battery life contributes negatively to user experience [21].

CHI'14, April 26–May 1, 2014, Toronto, ON, Canada.

http://dx.doi.org/10.1145/2556288.2557271

To automatically increase battery life or help users manage power consumption (a.k.a. *battery awareness*), numerous battery-saving applications have come to application markets [4, 15, 16, 25]. Although these applications can improve battery life, the automated solutions do not typically give users a direct indication of the concrete actions that make the battery last longer. Such applications, therefore, tend not to guide user behavior towards battery-saving choices.

There is prior work on the effect on user behavior of household energy awareness applications [1, 3] and mobile battery level indicators [20, 22]. However, we are not aware of any user behavior studies in the context of mobile battery awareness applications.

In this paper, we examine users of Carat, a community-based mobile battery-awareness application deployed worldwide to more than 670,000 devices. We conducted a survey of over 1,000 Carat users and analyze their responses along with data automatically gathered by Carat. Prior work on the Carat logs has shown not only that the application recommendations improve battery life (11% after 10 days and 40% after 90 days, on average), but that there is a positive correlation between the duration of using the application and the extent of the improvement [15]. One question we examine in this work is what distinguishes these long-term users from short-term users that might explain the difference in battery life improvement.

The contributions of this work are as follows:

- Elucidates the relationship between mobile battery awareness applications and user behavior;
- Examines two classes of users, distinguished by duration of use of the application, with distinct behaviors;
- Articulates lessons learned in the form of actionable guidelines for future battery awareness applications.

RELATED WORK

There is a growing body of literature on improving the battery life of mobile devices. The majority of this work consists of technical solutions, not intended for novice users [12, 16]. Other work provides suggestions or guidelines to users on how to reduce battery consumption [6, 14, 15]. There is another line of research on Human-Battery Interaction (HBI)

that focuses on user interaction with different battery indicators. Understanding the battery charging behavior and users' knowledge of power-saving features has been the subject of some HBI studies [2, 7, 20, 22]. In the domain of household energy awareness, there has been research on user behavior changes [1, 3]. We are not aware of any user behavior studies in the context of mobile battery awareness applications. As the goal of this research is to understand how mobile battery awareness applications change behavior, this section mainly considers work related to such applications, human-battery interaction, and behavior studies in the domain of household energy awareness.

Mobile Battery Awareness Applications

The primary goal of mobile battery awareness applications is to make the user aware of what consumes energy. The Android operating system has a built-in energy profiler that shows statistics about battery use on the device. This can be accessed from the battery option in the settings on most devices. Early consumer tools for energy awareness on smartphones include the Nokia Energy Profiler [4], which runs in the background, recording phone subsystem use, and later reports the energy use (in watts) over time. A more recent profiler is PowerTutor for Android [25], which shows energy use similarly to Android's built-in profiler but broken down by resource (e.g., CPU, WiFi, and the screen) and by category (e.g., by application or system component).

Carat is the first collaborative approach to mobile battery awareness, which allows it to perform diagnoses which would be impossible on a single device [15]. For example, although the tools discussed in this section are able to obtain accurate energy consumption profiles on a device, they cannot determine whether the amount of energy used by an application or device is normal. With a community of hundreds of thousands of client devices, Carat identified applications that consume abnormally large amounts of energy compared with other applications as well as instances of indiviual applications that consume abnormally large amounts of energy on only a subset of devices.

Human Battery Interaction (HBI)

Some work in HBI considers how users deal with limited mobile battery life. Banerjee et al. [2] studied phone battery-charging behavior and identified two categories of users: those who charge their phones regularly regardless of the remaining battery level and those who charge based on battery status indicators. Furthermore, there was usually a significant amount of power left in the battery when it was charged, even for users who charge based on the battery status indicator. There is also research on how battery-use feedback in mobile phones affects behavior [20, 22]. Studies revealed that some battery charging habits can reduce battery lifetime [7]. We extend these lines of inquiry by studying how mobile battery awareness applications affect behavior.

Behavior Change in Energy Awareness Applications

Literature on feedback for energy conservation spans several decades and includes work from several disciplines, including the behavioral sciences. This literature mostly considers domestic settings [18]. Although battery awareness applications address a different problem from domestic energy consumption feedback, there are similiarities. One key difference for mobile battery management is that power conservation is motivated by extending use time, while in domestic settings motivations are environmental or monetary.

Carat follows some of the key principles proposed in the energy consumption feedback literature; it provides actionable feedback, rewards users to keep them motivated, and avoids information overload. A solution tailored to individual users facilitates the job of persauding users to take the suggested actions, as each behavior has its own personalized reasons and constraints [1, 3].

Effective feedback should be real-time [3] and actionable, demonstrating a way to fill the gap between current actions and desired goal state [13]. The goal should always be clear to the user and be accompanied by instructions on how to achieve it.

Sustained involvement requires interfaces that evolve, rewarding improvements to keep the user motivated after the initial curiosity drops [11]. Recent work proposed a three-stage approach for feedback including the following: raise awareness, inform complex changes, and maintain sustainable routines [23].

There are a variety of other principles (e.g., format of feedback [9, 17, 19]), but the ones above are the most relevant for the case of battery awareness applications. Most available applications only provide simple feedback [23] ranging from power measures to monetary charges to carbon footprints. By themselves, such numbers do not suggest clear actions to take. Like recent energy consumption solutions that include contextually triggered advice [10], Carat provides concrete, actionable suggestions to users to improve battery life.

INTRODUCTION TO CARAT

Carat [15] uses a collaborative black-box method for diagnosing anomalies on mobile devices. Carat is an application on both the iOS and Android platforms.

The client application sends intermittent, coarse-grained measurements to a server. The server correlates running applications, device model, operating system, and other features with energy use. The system generates actions that the user could take to improve battery life. The amount of improvement error, and confidence of the suggestions given by Carat is presented to the user along with the actions. Carat has been installed on more than 670,000 devices.

On a single device, it is not possible to diagnose all types of abnormal energy use, because it could result from device or user-specific factors. A collaborative approach is required to diagnose energy bugs of this kind. Carat achieves this by using a community of devices.

Carat in Action

To walk through the features of Carat, we use the following scenario. John is a smartphone user with battery life issues. He starts by installing Carat on his phone. At first, when John

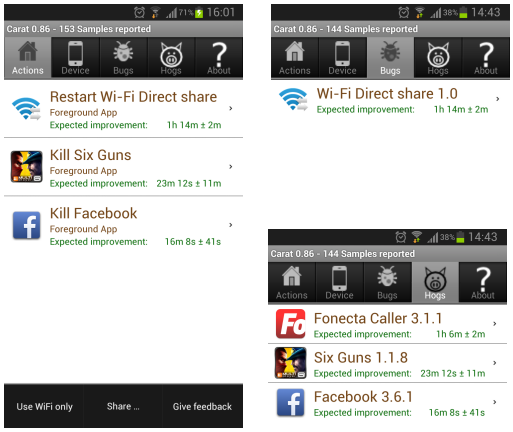


Figure 1. The Actions, Hogs, and Bugs screens

opens Carat, the system has no data on his device. However, he may see suggestions based on his currently running applications that are known anomalous energy consumers in the Carat community.

To calculate results for John's phone, Carat needs data from that phone. Carat application gathers intermittent measurements of the running applications, battery level, and other device features. When John opens Carat, it sends these data, referred to as samples, gathered so far to the server. John needs to open Carat regularly, preferably at least once a day, to receive personalized results as quickly as possible. He can see the number of sent samples in the top bar of the Carat application (Figure 1).

Actions

After about a week, John receives his first results. On the opening screen of Carat, the "Actions" tab, he sees suggestions given by Carat (Figure 1). There are usually two types of suggestions: "Kill application X" or "Restart application Y". Carat also shows how much the battery life is expected to increase if these applications were not used. If John wishes to kill or restart these applications, he can click on the corresponding item, and Carat will show a screen with instructions on how the application can be killed. Most of the time, John can kill an application just by clicking a button on this Carat screen, and Carat provides alternative instructions in case that fails. Sometimes, killing an application on Android does not succeed, because the application's background service restarts it right after it has been killed. Applications that behave like this can be *force killed* through the Task Manager, accessible from the same screen of Carat. However, doing this for applications required by the system can lead to instability, and often there is not much the user can do about them.

Bugs and Hogs

John can also see the applications mentioned in the "Actions" screen on the "Bugs" and "Hogs" tabs (See Figure 1). There might be other applications listed in those tabs as well, since actions are suggested only for the applications that are running on the users phone at that moment. On these tabs John sees lists of applications that he has been using after installing Carat that have been classified as *bugs* or *hogs*.

Hogs are applications that use more energy than an average application in the Carat community. Typically hogs require more energy for normal function; examples of this are VoIP, Internet radio, navigation, and camera applications. However, hogs can also result from a widespread problem with an application's energy use.

If the application is not a hog, it can still be a bug. A bug is an application that, for some reason, uses more battery than average on a specific device. For example, the Kindle application uses less energy, on average, than the average application, so it is not a hog. However, as reported in [15], some versions of the Kindle application had a bug which caused it to use more energy when connected via a mobile network. This made Kindle show up as a bug for Carat users who preferred mobile networks over WiFi. How hogs and bugs are calculated is described in detail by Oliner et al. [15].

The actions along with the hog and bug reports help John understand which applications are draining the battery faster than others, but also which of these applications are often running when the user opens the Carat application. He can use the actions screen to kill or restart running applications, and from hogs and bugs screens he can gain wider knowledge about applications that lower his battery lifetime and that of the other users in the Carat community.

J-Score and Other Information about Battery Life

After getting to know the energy efficiency of his applications better, John starts to get more interested in how well his battery lasts compared to other people. On the "Device" tab (see Figure 2) he sees a value called the J-Score. The J-Score tells him the percentage of devices in the Carat community that have a worse battery life than his phone. Underneath the J-Score is the expected active battery life calculated by Carat. This shows how long the battery would last if the device was used in the way that John has been using it since he started using Carat.

Other Features of Carat

John also sees some basic information about his phone on the "Device" tab: the operating system version, the device model, and information about the memory use of his phone. He can also look at a list of all the currently running applications by clicking the button "View Process List".

METHOD

We collected data from the existing Carat users by following two data collection methods: survey and system logs of Carat.

User Survey

Our first goal was to identify how different features of Carat affect user behavior. Therefore, we constructed a questionnaire and placed a link to it on the opening screen of Carat. The link was published to all Carat Android users. The survey was open for two weeks starting from August 12th, 2013.

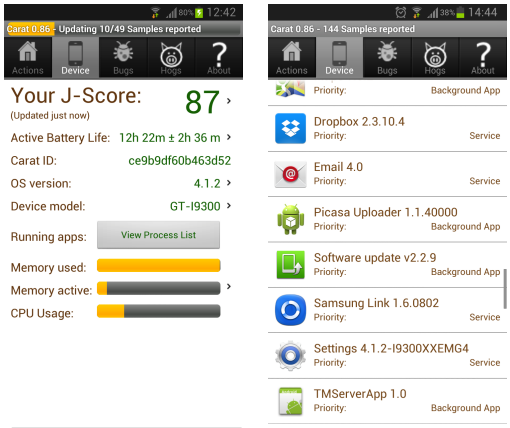


Figure 2. The Device screen and the process list

Survey Questionnaire

The questionnaire was composed of 16 questions plus optional free-text fields to express any additional comments regarding Carat application use experience and suggestions for improvement. All the questions were multiple choice, except for questions 8, 9, and 12, which were 7-point Likert scale. The full questionnaire with the multiple choice answer options can be found on the Carat website[1]. The following questions were asked in the questionnaire:

1. How long have you been using Carat?
2. What kind of device are you using now?
3. Do you use external batteries for this device?
4. How often do you charge the battery on this device?
5. On how many devices do you use Carat?
6. What is your main reason for using Carat?
7. Why did you choose Carat (instead of some other energy saving app)?
8. How well do you understand how Carat works?
9. Are you interested in knowing how Carat works?
10. What is the main reason for opening the Carat app?
11. Which of the following things do you do most times when you open Carat?
12. How often do you kill or restart an app when Carat suggests it?
13. What are the reasons why you don't kill an app when Carat suggests it?
14. How often have you opened Carat during the past month?
15. In what kind of situation do you usually open Carat?
16. In what ways has using Carat changed the way you use your device?

Response Statistics

A total of 1,140 valid responses were received from dozens of countries covering many of the regions with Carat users. Among the respondents 16% had been using Carat for over a year, while 40% had been using Carat for less than three months. Most of the respondents (93%) had been using Carat on their mobile devices while few of them had been using Carat on tablet devices. Around 26% of the respondents had been using Carat on more than one mobile device. 89% of the respondents were male, and the average age of respondents was 37 years. We are aware of the limitations of self-reporting, and we discuss them in the Limitations section.

[1] `http://carat.cs.berkeley.edu/chi/Carat-usage.html`

Samples	Reports
user id	average battery life (h)
date and time	hogs, the date they were found
battery level	bugs, the date they were found
running applications	list of applications the user has run

Table 1. Contents of Carat logs.

Carat Logs

In addition to the survey responses we also used automatically gathered Carat usage logs like Carat *samples* and *reports* of the users who answered the survey. The Carat application sends data to the servers in the form of samples. Each sample contains information about application use and battery life. After enough samples have been collected, Carat generates reports about users and applications. These reports are not available on the mobile client. These include details about the user's average battery life, the most battery-consuming applications that have been running on their device (hogs), and any applications that use more energy on their device than in the rest of the community (bugs). In this research we used these samples and reports to quantify user behavior. The contents of the two types of data used in this paper are detailed in Table 1. The logs give us important information, such as when a problematic application was reported to the user by Carat, and how that changed the behavior of the user in terms of running that application.

RESULTS

This section discusses the responses of the questionnaire and results of Carat log data analysis. To quantify differences and correlations in our results, we use two statistical tests. When comparing beginners and advanced users, we apply the Mann-Whitney U test; when discussing correlation, we use Kendall's tau (τ).

Beginners and Advanced Users

Prior work shows that there is a positive correlation between duration of Carat use and battery life [15]. In this paper we examine what are the reasons for this, and what features of Carat and user behaviors cause this positive correlation.

We found a significant positive correlation between the responses to *"How long have you been using Carat?"* and *"How well do you understand how Carat works?"* (seven-point Likert Scale where 1 = not at all, 7 = very well), $r_\tau(N = 1,140) = .13, p < .001$. There was also a significant positive correlation between the responses *"How long have you been using Carat?"* and *"On how many devices do you use Carat?"*, $r_\tau(N = 1,140) = .26, p < .001$. These results suggest that long-time Carat users believe that they

Definition of beginner	$U(Z)$	p	r
$<$ 1 month	81,027(-4.324)	$p < .01$	-.128
$<$ 3 months	132,180(-4.396)	$p < .01$	-.130
$<$ 6 months	136,352(-3.688)	$p < .01$	-.109
$<$ 1 year	71,478(-4.150)	$p < .01$	-.123

Table 2. Results of Mann-Whitney U test conducted among the two groups of Carat users (classified as beginners according to the given definition) on how well they understand how Carat works. Each classification scheme defines advanced users as those who have used longer than beginners.

Group Characteristics	Advanced Users	Beginners
Number of Respondents	689	451
Duration of Carat Use	>90 days	<90 days
Gender	male = 620 female = 58 other = 11	male = 389 female = 60 other = 2
Mean age in years	38	36

Table 3. Primary characteristics of the two groups of Carat users.

better understand how Carat works and have it installed on many devices.

We compared the responses to the question *"How long have you been using Carat?"* with the length of Carat usage logs from the device that was used to answer the questionnaire. The two are significantly correlated ($r_\tau(N = 1,072) = .357, p < .001$. The Carat logs underestimate actual duration of use because re-installation of the application or migration to a new device is recorded as a new user. The true correlation is therefore likely to be higher. In light of this limitation in the Carat log data, we use the questionnaire responses as a proxy for how long the respondents have used Carat.

Behavioral studies conducted with users of energy awareness applications have found that habits formed over three months are likely to stick with users [5]. We analyzed the survey responses to investigate the validity of this finding in the context of mobile battery awareness applications. First, we classified the respondents as beginners and advanced users by using each of the five options we gave them in the questionnaire (Less than a month, 1-3 months, 3-6 months, 6-12 months, and over year) as the threshold value. Next, for each classification we separately conducted a Mann-Whitney U test on how well they believe they understand how Carat works. A summary of the results is given in Table 2. According to the results, all the four classification schemes result in a significant difference between beginners and advanced users ($p < .01$). However, when we classify those who have been using Carat for less than three months as beginners, the relationship between the duration of use of Carat and how well they believe they understand it is stronger than in others ($r = -.130$). Based on these results and the importance of the three-month milestone in previous work [5], we use that as the classification threshold in this paper. Table 3 summarises the characteristics of these two groups.

Why Users Open Carat

In the survey, we asked the respondents to select from a list of options the main reason for opening Carat. We provided

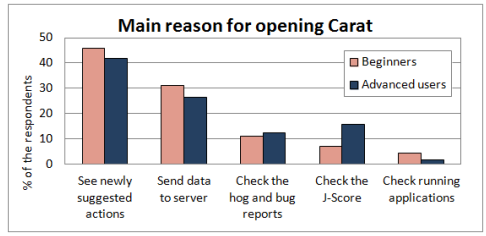


Figure 3. The main reason for opening Carat.

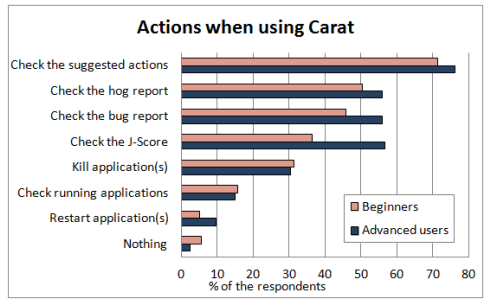


Figure 4. The actions performed when Carat is open.

the primary features of Carat as the options: send data to server, see newly suggested actions (kill or restart an application), check the reports (bugs and hogs), check the J-Score, and check running applications. Figure 3 summarizes the responses.

The majority of respondents (44%) mainly open Carat to see if any actions are suggested for them, and according to Figure 3 it is clear that nearly a similar proportion of beginners and advanced users have selected this reason. Group-wise analysis shows that for both beginners and advanced users, sending data to the server is the second-most-popular reason (27% of all the respondents selected this). Advanced users (16%) were more interested in checking the J-score than beginners (7%). The respondents were also asked which actions they perform most times when they open Carat (Figure 4). 71% of the beginners and 76% of the advanced users mentioned that they check the suggested actions. About half of all the users check the hog and bug reports, advanced users slightly more often than the beginners. The majority (57%) of the advanced users check the J-Score, but only 36% of the beginners are interested in it. A bit less than a third of the users kill applications most times they open Carat. The running applications are checked by 15% of the users, and applications are restarted by less than 10% of the users. Some users stated that they do nothing most of the time when they open Carat. The percentage of these users is higher among beginners (5.5% compared to 2.5% in the advanced users' group), probably because Carat does not give results to the user during the first week after installing Carat, so there is not much to do at that point.

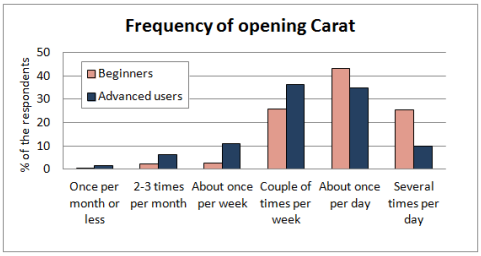


Figure 5. Summary of responses to the survey question "How often have you opened Carat during the past month".

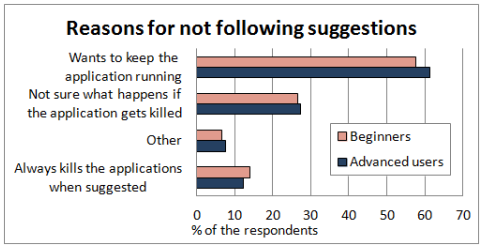


Figure 6. The reasons why users ignore suggestions to kill applications.

The primary reason for opening Carat and the most common actions performed after opening Carat do not vary much across beginners and advanced users. However, features like J-Score are more popular among advanced users. Based on these findings, we suggest that advanced users enjoy functions that support comparing against others in the Carat community.

How Often Users Open Carat

We asked the respondents to rate how often they have opened Carat during the past month. Beginners open Carat significantly more frequently than the advanced users. Figure 5 summarizes the responses to this question.

Beginners and advanced Carat users differ in their responses to the question *"How often have you opened Carat during the past month?"* (Figure 5 contains the options), ($U = 107,132, p < .001, r = .28$). Advanced Carat users had an average rank of 500 (Mdn = 4, SD = 1.08), while beginners had an average rank of 677 (Mdn = 5, SD =.95).

We infer that the suggestions provided by Carat are more useful to the beginners, and over time users learn to manage their battery without repeatedly checking Carat. These findings further suggest that the Carat use behavior changes over time, and the user's knowledge about how to improve battery life also grows with use of Carat.

Who Follows Suggestions and What They Gain

We asked the respondents to rate how often they kill or restart an application when Carat suggests it (seven-point Likert Scale where 1 = never and 7 = always). Most respondents follow application kill or restart suggestions (mean = 4.39) and beginners and advanced users follow Carat suggestions equally often. However, the users who claim to understand better how Carat works, charge their devices less often and follow Carat suggestions more often.

The difference between beginners (Mdn = 4, SD = 1.91) and advanced Carat users (Mdn = 5, SD = 1.76) on how often they follow Carat suggestions was not significant ($U = 155,203, p = .975, r = .0009$), suggesting that the two groups are similar in how often they kill or restart an application that Carat suggests.

There was a significant negative correlation between the responses to the questions "How well do you understand how Carat works?" and "How often do you charge the battery" ($r_\tau(N = 1,140) = -.084, p < .01$). Further, we also found a significant positive correlation between how often users kill or restart an application that Carat suggests, and how well user believe that they understand how Carat works ($r_\tau(N = 1,140) = .071, p < .01$). This suggests that even though the duration of use of Carat does not affect how often users follow Carat suggestions, how well the user understands how Carat works has an effect.

Statistical comparison between beginners and advanced Carat users on the percentage of battery life improvement (collected from Carat logs) after using Carat was significant ($U = 15,200, p < .05, r = -.102$). Advanced Carat users had an average rank of 227 (Mdn = .011, SD = 10.2), while beginners had an average rank of 196 (Mdn = -.53, SD = 10.1). In agreement with prior work [15], we found that duration of use correlates positively with battery life improvement.

All these results suggest that Carat fosters learning, and as a result of that users learn to manage their battery better with long-term use. This encourages the users to stick with Carat. We conclude that energy awareness applications should make the logic behind their suggestions understandable to the users in order to support learning, and encourage them to follow the suggestions and use the application for long.

Why Users Ignore Suggestions

The users of energy awareness applications do not always follow the suggestions provided to them. In order to find why Carat users sometimes ignore the suggestions, we asked the respondents to select from a list of options all the reasons for not killing an application suggested by Carat. The options were: I want to keep it running, I'm not sure what happens if I kill it, I always kill the application when suggested, and a free text field to provide other options. Figure 6 provides a summary of responses. The main reason for ignoring the suggestions to kill applications is that the user wants to keep that application running (58% of the beginners and 61% of the advanced users) regardless of its high power consumption. Some of the respondents provided further justifications for this option in the free text field. According to them, one of the most common reasons that eight beginners and 18 advanced users stated was that some applications cannot be killed. Thirteen advanced users and two beginners stated that they check the estimated battery improvement provided by Carat, and if it's too low they do not kill the suggested application. Seven

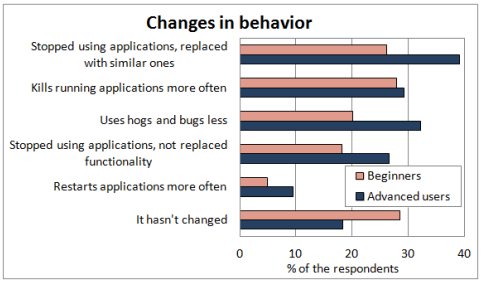


Figure 7. Summary of responses to the survey question "In what ways using Carat has changed the way you use your device?".

advanced users and one beginner mentioned that sometimes Carat suggests to kill system applications. Eight beginners also stated that they have heard that killing applications in Android is bad.

We conclude that suggestions to kill system applications and regularly used applications are not very useful to the user. However, the estimated battery life improvement number provides additional information for the user to decide whether to kill an application or not.

How Carat Changes Device Use

To understand how Carat has influenced the mobile device use behavior, we asked the respondents to select all the relevant options from a list of user behavior changes that we expected Carat to cause. Responses revealed that Carat has caused behavioral changes especially in advanced Carat users. Figure 7 provides a graphical illustration of the list of given options and the percentage of users selected each option.

Carat did not affect the mobile device use behavior of 29% of the beginners. However, 39% of advanced users agreed that Carat has made them stop using some applications and replace them with similar ones. The second most common change that Carat has caused in 28% of beginners and 29% of advanced users is killing running applications more often. Advanced Carat users agreed on all the behavioral changes more than the beginners.

These results suggest that Carat has a bigger impact on the mobile device use behavior of advanced users, and it takes time for new users to adapt these new habits. This further explains why initial performance analysis indicated that the battery life of Carat users improves gradually over time [15].

Who Kills Bugs, Hogs, and Other Applications

From the Carat logs, we learned that beginners reduced the use of 64.3% of their hogs and bugs on average when they were first reported. All hogs were reduced by 36.5% and bugs by 23% on average. Advanced users reduced the use of 67.2% of their hogs and bugs, hogs by 46% and bugs by 30%. Table 4 shows these reduction ratios.

Furthermore, we compared how much beginners and advanced Carat users reduce the use of hogs and bugs. The difference for hogs was significant ($U = 58,369, p < .001, r = .17$). Advanced Carat users had an average rank of 427 (Mdn = 45, SD = 23) while beginners had an average rank of 345 (Mdn = 36, SD = 25). These results indicate that advanced users reduce the use of hogs significantly more than the beginners. This can be one of the reasons why advanced users improve battery life better than the beginners. However, the test for bugs was not significant $U = 12,995, p = .189, r = .07$, indicating that both the beginners (Mdn = 27 , SD = 38) and advanced users (Mdn = 33, SD = 39) have equally reduced the use of bugs. We also calculated the reduction in use of other applications that have not been reported as hogs or bugs. Here we considered actions, such as starting to use new applications, abandoning old ones, and killing applications for battery saving. Beginners have reduced use of these other applications by 11.51% and advanced users by 24.14%. Since the percentage of reduction in use of other applications was normally distributed, we conducted an independent T-test on this data and found that the advanced users (M=17.9%, SD=1.99) have reduced the use of other applications significantly more than the beginners (M=5.81%, SD=1.86); t(851)=-8.88, p < .001. These results along with our previous findings that show that advanced users open Carat less often, yet have better battery life suggest that advanced users have learned to better manage their battery with less help from Carat. Given below is how bugs, hogs, and other application reduction percentages are calculated.

Calculation of Application Use Reductions

We examined the Carat samples and the Carat log reports of the survey respondents. For each user, we obtained the total number of samples u_t they reported to Carat. For the first hog or bug report of each application z for each user u, we split u_t into the samples before the report u_b and after u_a. We took the subset of samples that contained z, before z_{ub} and after z_{ua} the report. Finally, we only considered the samples before the report from the point that z was first run t_{uz1} by u to avoid diluting the ratio: $u_{z1} = u_b|\text{time} >= t_{uz1}$. Then we compared the ratio of running the application before the report, to after it, and obtained the reduction ratio r:

$$r = 1 - \frac{z_{ua}/u_a}{z_{ub}/u_{z1}}. \quad (1)$$

Note that if the user increased the ratio of running the application, then $r < 0$. We then calculated the averages of decreasing application use for all u in each category, and all z for all the reports that we had obtained from Carat:

$$a = \frac{\sum_u^n \sum_z^m r}{n \times m}. \quad (2)$$

The Respondents' Comments on Carat

We asked the respondents to comment on what they specifically like about Carat, and provide suggestions for improvement. This was an optional part of the survey. Hence only 20.6% of the respondents provided comments. The advanced

Category	% of Hogs/Bugs affected	Hog % reduced	Bug % reduced
Beginners	64.3	36.50	23.48
Advanced users	67.2	46.35	30.12

Table 4. The percentage of Hogs/Bugs that the users reduced the use of, and the reduction percentages, the first time a user gets a report on them.

users provided more suggestions for improvement than beginners, which is natural since they have more experience with Carat.

Most Liked Features of Carat

We received comments about preferable features of Carat from 6.4% of the beginners and 8.6% of the advanced users. Most of the comments were not addressing any specific features, but rather stating general interest in Carat.

The hog and bug reports were positively acknowledged by 12 beginners and 13 advanced users in their comments: *"I really like how [Carat] tells you about buggy [applications] and [...] hogs."*

23 advanced users stated that they like the J-Score the most: *"The J-Score is a great way for comparing battery life with other devices."* However, only four beginners expressed their interest in J-Score. This is in line with our previous findings about the advanced users being more interested in the J-Score.

14 advanced users and 7 beginners admired non-functional features of Carat, such as reliability, usability, and low battery consumption: *"[Carat] just works without being a hog itself."*, *"[Carat] works very well and is very simple to use."* Respondents also mentioned using Carat because it does not kill applications by itself but gives control to the user.

Another feature that was mentioned in many comments was the actions tab, and the fact that other applications can be killed directly through Carat. This also gave users information about applications that restart right after killing them: *"I have found it useful to see which [applications] [...] are constantly restarted by built-in [...] software."*

Suggestions for Improvement

8.5% of the beginners and 18% of the advanced users made suggestions for improvement. Figure 8 provides a summary of these suggestions. Many of the suggestions were about additional features such as automatic collection of samples, but a significant number of respondents also requested more information about current features. 27 advanced users and five beginners suggested that Carat should send samples automatically or show periodic reminders to open Carat often enough. We also received comments requesting more information about hogs and bugs. Among them we identified three types of problems concerning the actions suggested by Carat: insufficient information about applications reported as hogs/bugs, system applications are suggested for killing, and no solution for applications that reopen immediately after killing. Three beginners and 17 advanced users stated that they would like to have more information about applications that are suggested to be killed, such as what it does, and suggestions for substitute applications. 12 advanced users stated that Carat is

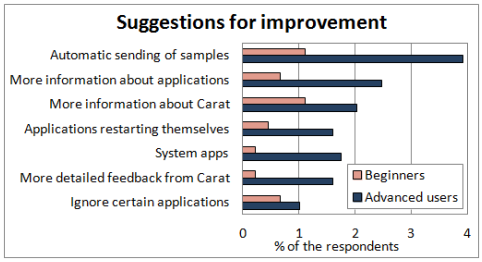


Figure 8. Types of suggestions for improvement

suggesting them to kill system applications: *"[...] sometimes [the applications suggested for killing] seem like system [applications] or important services."*. Respondents also stated that they would like Carat to detect applications that reopen immediately after killing, and make alternative suggestions for them. Furthermore, seven advanced users and three beginners stated that sometimes Carat suggests them to kill applications that they use regularly. They prefer a way to hide hogs and bugs that they want to use: *"Having a way to ignore some [applications] would be great."*

Five beginners and 14 advanced users commented that they need more information about how Carat works. Beginners stated more directly that they do not understand how Carat works. However, advanced users stated that they misinterpreted some features of Carat, or the feature that the user needed more information about was often specified: *"[I] wish I understood what expected improvement means."*

LIMITATIONS

The limitations of self-reporting are well-known. Since the survey respondents were a group of self-opted volunteers among all Carat users, this group might be more interested in features of Carat than the other users. In addition, subjects may, intentionally or not, provide inaccurate or imprecise responses. To address this, we combined Carat log data with user-reported data where possible. However, some aspects cannot be corroborated with Carat log data, such as gender and understanding how Carat works. Since the majority of the respondents were male (89%) the results may not generalize so well to female users and we have no ability to compensate for potential gender biases or incorrect user beliefs on their understanding of how Carat works. For the purpose of our discussion we assume that the gender of respondents is not correlated with the features of interest such as how much their battery life improved.

There could be other external factors such as users' long term experience with smart phones that could have influenced the battery management skills of users. However, we assume the duration of Carat use is the dominant factor, because previous Carat studies showed that not only does user battery life improve over time, but that this improvement is much stronger for users who receive suggestions from Carat compared with those who do not [15], and also literature [5] suggests that the users form habits with long-term use of energy awareness applications.

Our results track groups, not individual users, and their behavior. In future work, we will conduct longitudinal studies with Carat users and analyze their behavior in more detail.

DISCUSSION AND CONCLUSIONS

We discovered features of Carat that influence user behavior and how behavior changes with long-term use of the application. The findings deepen our knowledge on how to improve community-based battery awareness applications to better support both new and long-term users.

We conducted a survey with existing users of the application, and analyzed their interaction logs from Carat. With the help of these quantitative and qualitative data, we compared the behavior of two types of users, beginners and advanced, to better understand why the latter group enjoys a greater improvement to battery life. Our results revealed that advanced users open Carat less frequently than beginners. However, Carat has considerably changed the mobile device use behavior of advanced users. They have stopped using some applications and replaced them with alternatives, have gained better battery life, charge their devices less frequently, kill reported hogs and bugs more often, and have learned to better manage their battery without the help of Carat. These findings suggest that Carat has changed user behavior while helping users learn to identify applications that drain the battery quickly. Building on these observations, we propose a set of guidelines applicable to the design of battery-awareness applications.

First Guideline: Show Your Work

Carat has succeeded in changing behavior by combining crowdsourcing with explicit instructions that are missing in many similar battery awareness applications [8, 16]. Carat provides explicit information about which applications are draining the battery abnormally quickly through its action list, and the bug and hog reports help the user understand how these applications are affecting the broader community. These are primary features of Carat that enhance user knowledge. Furthermore, information about expected battery life improvement helps users learn how killing an application actually affects the battery life. Our findings indicate that advanced users get into the habit of checking expected battery life improvement before killing applications, and that such features foster learning about mobile battery life and application behavior. In household energy awareness systems, it was recommended to provide feedback to support learning [5]. Our results also suggest that users are not interested in blindly following instructions, but seem to follow Carat suggestions more often when they understand how it works. This understanding helps the user trust the recommendations and possibly learn enough to make similar diagnoses on their own in the future. According to these findings we propose our first guideline: *Expose to the user not just recommendations, but also the reasoning or data behind them.*

Second Guideline: Retain Long-Term Users

If the goal of battery awareness applications is to improve users' knowledge of the device and increase battery life, then prolonged use should result in increasingly better battery life. In Carat, this effect is amplified by increasingly accurate recommendations as Carat learns more about the user's device. However, our findings also suggest that there is a tendency for long-term users to leave Carat once they have learned to manage their battery without the help of the application. Tailoring features for different types of users has been a challenge in domestic energy awareness research [1, 3]. Community-based mobile battery awareness systems that learn from their users should also be tailored to retain long-term users. The J-Score feature in Carat tries to achieve this retention through community engagement. Advanced Carat users are more interested in the J-Score, and the competitive environment it creates. According to these results, we propose our second guideline: *Tailor community-based battery awareness applications to retain long-term users.*

Third Guideline: Give clear, action-oriented instructions for improving battery life

Providing effective feedback on resource consumption is a key challenge in household energy awareness systems [24]. It is important to give feedback in way that is easy for users to grasp; the instructions should be unambiguous and action-oriented. As shown in Figure 4, the most popular feature of Carat was the "actions" tab. These suggested actions were more popular than the hogs or bugs, even though they simply tell users to "kill" or restart running applications on the hogs and bugs lists. However, in our study we found that the term "killing" an application was misinterpreted by some users, since they feared that killing would result in data loss. The term was chosen to represent permanently closing an application and keeping it closed. Unfortunately, some applications automatically restart when killed, for example Facebook on Android restarts unless "Force Closed" through the Application Manager. Based on these findings we propose our third guideline: *Take into account the audience when formulating feedback to convey precisely what is intended. Provide the user with clear, action-oriented instructions for improving battery life.*

Fourth Guideline: Distinguish System Components

System components pose a problem for Carat, as they are sometimes difficult to distinguish from third-party applications and require different treatment with respect to kill/restart recommendations. Carat maintains a list of system applications in order to mitigate this problem. However, with new versions of mobile operating systems and custom Android versions, maintaining an up-to-date list is a difficult task. Android provides a flag that indicates whether an application is part of the pre-installed image on a device, yet many service providers include applications that can be safely killed such as Facebook and Twitter in the pre-installed applications. This problem can be addressed through crowdsourcing by allowing users to flag suspected system applications. We propose our fourth guideline based on this example: *Distinguish system components from third-party applications when making diagnoses and recommendations.*

The findings presented in this paper provide suggestions for the improvement of mobile battery awareness applications. The guidelines above target community-based mobile battery

awareness applications. Single-device applications can take advantage of all but the second guideline.

REFERENCES

1. Abrahamse, W., Steg, L., Vlek, C., and Rothengatter, T. The effect of tailored information, goal setting, and tailored feedback on household energy use, energy-related behaviors, and behavioral antecedents. *J. Environmental Psych. 27*, 4 (2007), 265–276.
2. Banerjee, N., Rahmati, A., Corner, M. D., Rollins, S., and Zhong, L. Users and batteries: Interactions and adaptive energy management in mobile systems. In *Proc. UbiComp 2007*. Springer, 2007, 217–234.
3. Chetty, M., Tran, D., and Grinter, R. E. Getting to green: understanding resource consumption in the home. In *Proc. UbiComp*, ACM (2008), 242–251.
4. Creus, G., and Kuulusa, M. Optimizing mobile software with built-in power profiling. In *Proc. Mobile Phone Programming*, F. Fitzek and F. Reichert, Eds. Springer Netherlands, 2007, 449–462.
5. Darby, S. The effectiveness of feedback on energy consumption. *A Review for DEFRA of the Literature on Metering, Billing and direct Displays 486* (2006), 1–21.
6. Datta, S. K., Bonnet, C., and Nikaein, N. Minimizing energy expenditure in smart devices. In *Proc. ICT*, IEEE (2013), 712–717.
7. Ferreira, D., Dey, A. K., and Kostakos, V. Understanding human-smartphone concerns: a study of battery life. In *Proc. Pervasive Computing*. Springer, 2011, 19–33.
8. Ferreira, D., Ferreira, E., Goncalves, J., Kostakos, V., and Dey, A. K. Revisiting human-battery interaction with an interactive battery interface. In *Proc. UbiComp*, ACM (2013).
9. Froehlich, J., Findlater, L., and Landay, J. The design of eco-feedback technology. In *Proc. CHI*, ACM (2010), 1999–2008.
10. Gamberini, L., Spagnolli, A., Corradi, N., Jacucci, G., Tusa, G., Mikkola, T., Zamboni, L., and Hoggan, E. Tailoring feedback to users actions in a persuasive game for household electricity conservation. In *Persuasive Technology. Design for Health and Safety*. Springer, 2012, 100–111.
11. Henryson, J., Håkansson, T., and Pyrko, J. Energy efficiency in buildings through information–swedish perspective. *J. Energy Policy 28*, 3 (2000), 169–180.
12. Liu, X., Shenoy, P., and Corner, M. Chameleon: application level power management with performance isolation. In *Proc. Intl. Conf. on Multimedia*, ACM (2005), 839–848.
13. Midden, C. J., Meter, J. F., Weenig, M. H., and Zieverink, H. J. Using feedback, reinforcement and information to reduce energy consumption in households: A field-experiment. *J. Econ. Psych. 3*, 1 (1983), 65–86.
14. Oliner, A., Iyer, A., Lagerspetz, E., Tarkoma, S., and Stoica, I. Collaborative energy debugging for mobile devices. *Proc. USENIX HotDep* (2012).
15. Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. Carat: Collaborative energy diagnosis for mobile devices. In *Proc. SenSys* (2013).
16. Pathak, A., Hu, Y. C., and Zhang, M. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proc. European Conf. on Computer Systems*, ACM (2012), 29–42.
17. Pierce, J., Fan, C., Lomas, D., Marcu, G., and Paulos, E. Some consideration on the (in) effectiveness of residential energy feedback systems. In *Proc. DIS*, ACM (2010), 244–247.
18. Pierce, J., and Paulos, E. Beyond energy monitors: interaction, energy, and emerging energy systems. In *Proc. CHI*, ACM (2012), 665–674.
19. Pierce, J., Schiano, D. J., and Paulos, E. Home, habits, and energy: examining domestic interactions and energy consumption. In *Proc. CHI*, ACM (2010), 1985–1994.
20. Rahmati, A., Qian, A., and Zhong, L. Understanding human-battery interaction on mobile phones. In *Proc. MobileHCI*, ACM (2007), 265–272.
21. Rahmati, A., Tossell, C., Shepard, C., Kortum, P., and Zhong, L. Exploring iphone usage: The influence of socioeconomic differences on smartphone adoption, usage and usability. In *Proc. MobileHCI*, ACM (2012), 11–20.
22. Rahmati, A., and Zhong, L. Human–battery interaction on mobile phones. *J. Pervasive and Mobile Comp. 5*, 5 (2009), 465–477.
23. Riche, Y., Dodge, J., and Metoyer, R. A. Studying always-on electricity feedback in the home. In *Proc. CHI*, ACM (2010), 1995–1998.
24. Rodgers, J., and Bartram, L. Exploring ambient and artistic visualization for residential energy use feedback. *J. IEEE Trans. Visualization and Comp. Graphics 17*, 12 (2011), 2489–2497.
25. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis*, ACM (2010), 105–114.

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FI-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports are available on the e-thesis site of the University of Helsinki.

A-2009-1 K. Hätönen: Data mining for telecommunications network log analysis. 153 pp. (Ph.D. Thesis)

A-2009-2 T. Silander: The Most Probable Bayesian Network and Beyond. 50+59 pp. (Ph.D. Thesis)

A-2009-3 K. Laasonen: Mining Cell Transition Data. 148 pp. (Ph.D. Thesis)

A-2009-4 P. Miettinen: Matrix Decomposition Methods for Data Mining: Computational Complexity and Algorithms. 164+6 pp. (Ph.D. Thesis)

A-2009-5 J. Suomela: Optimisation Problems in Wireless Sensor Networks: Local Algorithms and Local Graphs. 106+96 pp. (Ph.D. Thesis)

A-2009-6 U. Köster: A Probabilistic Approach to the Primary Visual Cortex. 168 pp. (Ph.D. Thesis)

A-2009-7 P. Nurmi: Identifying Meaningful Places. 83 pp. (Ph.D. Thesis)

A-2009-8 J. Makkonen: Semantic Classes in Topic Detection and Tracking. 155 pp. (Ph.D. Thesis)

A-2009-9 P. Rastas: Computational Techniques for Haplotype Inference and for Local Alignment Significance. 64+50 pp. (Ph.D. Thesis)

A-2009-10 T. Mononen: Computing the Stochastic Complexity of Simple Probabilistic Graphical Models. 60+46 pp. (Ph.D. Thesis)

A-2009-11 P. Kontkanen: Computationally Effcient Methods for MDL-Optimal Density Estimation and Data Clustering. 75+64 pp. (Ph.D. Thesis)

A-2010-1 M. Lukk: Construction of a global map of human gene expression - the process, tools and analysis. 120 pp. (Ph.D. Thesis)

A-2010-2 W. Hämäläinen: Efficient search for statistically significant dependency rules in binary data. 163 pp. (Ph.D. Thesis)

A-2010-3 J. Kollin: Computational Methods for Detecting Large-Scale Chromosome Rearrangements in SNP Data. 197 pp. (Ph.D. Thesis)

A-2010-4 E. Pitkänen: Computational Methods for Reconstruction and Analysis of Genome-Scale Metabolic Networks. 115+88 pp. (Ph.D. Thesis)

A-2010-5 A. Lukyanenko: Multi-User Resource-Sharing Problem for the Internet. 168 pp. (Ph.D. Thesis)

A-2010-6 L. Daniel: Cross-layer Assisted TCP Algorithms for Vertical Handoff. 84+72 pp. (Ph.D. Thesis)

A-2011-1 A. Tripathi: Data Fusion and Matching by Maximizing Statistical Dependencies. 89+109 pp. (Ph.D. Thesis)

A-2011-2 E. Junttila: Patterns in Permuted Binary Matrices. 155 pp. (Ph.D. Thesis)

A-2011-3 P. Hintsanen: Simulation and Graph Mining Tools for Improving Gene Mapping Efficiency. 136 pp. (Ph.D. Thesis)

A-2011-4 M. Ikonen: Lean Thinking in Software Development: Impacts of Kanban on Projects. 104+90 pp. (Ph.D. Thesis)

A-2012-1 P. Parviainen: Algorithms for Exact Structure Discovery in Bayesian Networks. 132 pp. (Ph.D. Thesis)

A-2012-2 J. Wessman: Mixture Model Clustering in the Analysis of Complex Diseases. 118 pp. (Ph.D. Thesis)

A-2012-3 P. Pöyhönen: Access Selection Methods in Cooperative Multi-operator Environments to Improve End-user and Operator Satisfaction. 211 pp. (Ph.D. Thesis)

A-2012-4 S. Ruohomaa: The Effect of Reputation on Trust Decisions in Inter-enterprise Collaborations. 214+44 pp. (Ph.D. Thesis)

A-2012-5 J. Sirén: Compressed Full-Text Indexes for Highly Repetitive Collections. 97+63 pp. (Ph.D. Thesis)

A-2012-6 F. Zhou: Methods for Network Abstraction. 48+71 pp. (Ph.D. Thesis)

A-2012-7 N. Välimäki: Applications of Compressed Data Structures on Sequences and Structured Data. 73+94 pp. (Ph.D. Thesis)

A-2012-8 S. Varjonen: Secure Connectivity With Persistent Identities. 139 pp. (Ph.D. Thesis)

A-2012-9 M. Heinonen: Computational Methods for Small Molecules. 110+68 pp. (Ph.D. Thesis)

A-2013-1 M. Timonen: Term Weighting in Short Documents for Document Categorization, Keyword Extraction and Query Expansion. 53+62 pp. (Ph.D. Thesis)

A-2013-2 H. Wettig: Probabilistic, Information-Theoretic Models for Etymological Alignment. 130+62 pp. (Ph.D. Thesis)

A-2013-3 T. Ruokolainen: A Model-Driven Approach to Service Ecosystem Engineering. 232 pp. (Ph.D. Thesis)

A-2013-4 A. Hyttinen: Discovering Causal Relations in the Presence of Latent Confounders. 107+138 pp. (Ph.D. Thesis)

A-2013-5 S. Eloranta: Dynamic Aspects of Knowledge Bases. 123 pp. (Ph.D. Thesis)

A-2013-6 M. Apiola: Creativity-Supporting Learning Environments: Two Case Studies on Teaching Programming. 62+83 pp. (Ph.D. Thesis)

A-2013-7 T. Polishchuk: Enabling Multipath and Multicast Data Transmission in Legacy and Future Interenet. 72+51 pp. (Ph.D. Thesis)

A-2013-8 P. Luosto: Normalized Maximum Likelihood Methods for Clustering and Density Estimation. 67+67 pp. (Ph.D. Thesis)

A-2013-9 L. Eronen: Computational Methods for Augmenting Association-based Gene Mapping. 84+93 pp. (Ph.D. Thesis)

A-2013-10 D. Entner: Causal Structure Learning and Effect Identification in Linear Non-Gaussian Models and Beyond. 79+113 pp. (Ph.D. Thesis)

A-2013-11 E. Galbrun: Methods for Redescription Mining. 72+77 pp. (Ph.D. Thesis)

A-2013-12 M. Pervilä: Data Center Energy Retrofits. 52+46 pp. (Ph.D. Thesis)

A-2013-13 P. Pohjalainen: Self-Organizing Software Architectures. 114+71 pp. (Ph.D. Thesis)

A-2014-1 J. Korhonen: Graph and Hypergraph Decompositions for Exact Algorithms. 62+66 pp. (Ph.D. Thesis)

A-2014-2 J. Paalasmaa: Monitoring Sleep with Force Sensor Measurement. 59+47 pp. (Ph.D. Thesis)

A-2014-3 L. Langohr: Methods for Finding Interesting Nodes in Weighted Graphs. 70+54 pp. (Ph.D. Thesis)

A-2014-4 S. Bhattacharya: Continuous Context Inference on Mobile Platforms. 94+67 pp. (Ph.D. Thesis)